DIMACS Technical Report 2007-10

Time-Decaying Aggregates in Out-of-order Streams

by

Graham Cormode[1]
AT&T Labs–Research
graham@research.att.com

Flip Korn[2]
AT&T Labs–Research
flip@research.att.com

Srikanta Tirthapura
Iowa State University
snt@iastate.edu

[1]DIMACS Permanent Member
[2]DIMACS Permanent Member

ABSTRACT

Processing large data streams is now a major topic in data management. The data involved can be truly massive, and the required analyses complex. In a stream of sequential events such as stock feeds, sensor readings, or IP traffic measurements, tuples pertaining to recent events are typically more important than older ones. This can be formalized via decay functions, which assign weights based on age. Decay functions such as sliding windows and exponential decay have been well studied under the assumption of well-ordered arrivals, i.e., data arrives with increasing time stamps. However, data quality issues are prevalent in massive streams (due to network asynchrony and delays etc.), and correct arrival order is not guaranteed.

We focus on the computation of decayed aggregates such as range queries, quantiles, and heavy hitters on out-of-order streams, where elements do not necessarily arrive in increasing order of timestamps. We give the first deterministic algorithms for approximating these aggregates under the sliding window, exponential and polynomial decay functions. Our techniques are based on extending existing data stream summaries, such as the q-digest. We show that the overhead for allowing out-of-order arrivals is small, compared to the case of well-ordered arrivals. Our experiments confirm that our methods can be applied in practice, and show how the specific decay function impacts performance.

1 Introduction

The challenge of observing, processing and analyzing massive streams of data is now ensconced as a major topic within data management. The rapid growth in data volumes from applications such as networking, scientific experiments and automated processes continues to surpass our ability to store and process using traditional means. Consequently, a new generation of systems and algorithms has been developed, under the banner of "data streaming". Several axioms are inherent in this world: the value in the data lies not in simple calculations on specific predetermined subsets but rather in potentially complex aggregates computed over the bulk of it; queries over the data must be answered quickly, often continuously; and although the data may be archived onto slow media, computation for analysis must preferably be done online as the stream is observed in whatever order it arrives.

In contrast with a stored database, events in a data stream that have occurred recently are usually more significant than those in the distant past. This is typically handled through *decay functions* that assign greater weight to more recent elements in the computation of aggregates. Various models have been proposed for how to set these weights; for instance, the *sliding window* model considers only the most recent events and ignores the rest [14, 16]. Observe that in a typical network monitoring scenario, the number of events within, say, a 24 hour window, can be many millions or even billions [24, 5]. So the same challenges of data size and volume hold even if we are computing aggregates over a sliding window. More generally, one can design arbitrary schemes that allocate weights to observed data as a function of the "age" of the data. Another popular decay model is *exponential decay*, where the weight of data decreases exponentially with the age—the popularity of this model is due in part to the relative simplicity of algorithms to implement exponential decay. In many situations a polynomially decreasing decay function may be more appropriate [9].

A significant challenge in processing data streams transmitted across a network is to cope with the network asynchrony, and hence the imperfect ordering of data within a stream. Often, massive streams occur as the aggregation of multiple feeds from different sources. For example, a data stream observed by the "sink" node in a sensor network is formed by the union of multiple streams, each stream consisting of the observations generated by an individual sensor node. The different streams could be arbitrarily interleaved at the sink node (or at the intermediate nodes of the sensor network), due to the varying delays, retransmissions or intermittent connectivity in sensor networks. So if each sensor observation is tagged with a timestamp indicating the time of observation, the data stream observed by the sink may not necessarily be in the order of increasing timestamps, even though the observations due to an individual sensor node may be in the order of increasing timestamps. We refer to such a stream, where data is not ordered according to timestamps, as an "out-of-order" stream.

In an out-of-order stream, the notion of "recency" is defined using timestamps of the data, rather than the order of arrival. For example, in computing aggregates over a "sliding window" of W recent elements on an out-of-order stream, we should consider the elements with the W *greatest timestamps*. In contrast, the traditional definition of a sliding window [14, 16]

computes on the suffix (of size W) of the *most recently received* elements.

Various approaches have been suggested to cope with small ordering discrepancies within a stream: using buffers to reorder [1], "punctuation" of the stream to indicate no further events from a particular time window are expected [28], or load shedding when no other option is left [5]. However, such approaches only apply to certain special cases, or require too much overhead to put the stream in sorted order. Therefore, we need techniques to summarize and analyze massive streaming data that are resilient to data arriving in arbitrary orders, yet allow different decay functions to be applied. Moreover, we need general approaches which can allow the incorporation of decayed aggregates into streaming systems, and automated handling of different decay functions from high level specifications.

The main contributions of this paper are as follows:

- We present efficient algorithms for out-of-order streams that can answer queries for key aggregates such as *range-queries*, *quantiles* and *heavy-hitters (frequent elements)* under different decay functions such as sliding window decay, exponential decay and polynomial decay.
- Our algorithms provide deterministic approximation guarantees for the quality of the solutions returned. For example, in the case of quantiles, the summary returns the ϵ-approximate (weighted) ϕ-quantile of the data, which is an element whose (weighted) relative rank in the data is guaranteed to be between $(\phi - \epsilon)$ and $(\phi + \epsilon)$. We provide similar guarantees for range-queries and heavy hitters, too. These guarantees hold independent of the amount of disorder in the streams. We show asymptotic space and time bounds for our algorithms which are sublinear in the input size.
- We provide a comprehensive experimental evaluation of the algorithms discussed. We observe that for some decay functions the cost of handling decay and disorder is negligible; and for others the throughput is still acceptable.

Outline. We first review preliminaries in Section 2. We give a solution for tracking Sliding Window Decay for aggregates in Section 3. Techniques for exponential decay functions are given in Section 4. We outline two approaches to general decay functions in Section 5: one through a reduction to sliding windows, the second using the structure of the decay function. We give our experimental results in Section 6, and then discuss extensions and conclusions.

2 Preliminaries

2.1 Streaming Model

Definition 1. *A data stream is an (unbounded) sequence of tuples $e_i = \langle x_i, t_i \rangle$, where x_i is the identifier of the item (the key) and t_i the timestamp.*

For example, consider observing a stream of (IP) network packets. There are several ways to abstract a data stream from this: we could have x_i be the destination address, and t_i the time of observation of the packet; or, x_i be the concatenation of the source and destination addresses, and t_i be a timestamp encoded in the body of the packet indicating the time of origin (e.g. by a VoIP application). All the methods we discuss can naturally and immediately handle the case of *weighted updates*, where each tuple arrives with an associated weight w_i (e.g., the size of a packet in bytes). Effectively, the algorithms treat all unweighted updates as updates with weight 1, and so can replace this value with an arbitrary weight. For simplicity, we do not explicitly include the weighted case, since the details are mostly straightforward. So we do not discuss further the mapping of the raw data to $\langle x_i, t_i \rangle$ tuples, but instead assume that the tuples can be easily extracted.

The "current time" is denoted by the variable t. It is assumed that all times are non-negative integer values. Since we consider out-of-order arrivals, the timestamp t_i is completely decoupled from the time at which the tuple is observed. Thus it is possible that $i < j$, so that $e_i = \langle x_i, t_i \rangle$ is received earlier than $e_j = \langle x_j, t_j \rangle$, but $t_i > t_j$ so that e_i is in fact a more recent observation than e_j. Note that it is possible that there are many items in the stream with the same timestamp, or none.

It is important to distinguish between the timestamp of an item, t_i, and the *age* of an item: the age of item $\langle x_i, t_i \rangle$ is $t - t_i$. While the age of an item is constantly changing with the current time, its timestamp remains the same. To emphasize this difference, we will indicate times with t_i, T etc., and ages with a_i, A, etc. There are two potential variations here: the first where the age of an item is directly computed from its timestamp, and a second where the age of an item is the number of items seen with more recent timestamps. In this paper we focus on time-based semantics, and note that many of our techniques apply to tuple-based semantics.[1]

2.2 Decay Functions

We next formalize the notion of a decay function, which takes the age of an item, and returns a weight for this item. We define a function $g(a)$ to be a *decay function* if it satisfies the following properties:

1. $g(0) = 1$ and $0 \leq g(a) \leq 1$ for all $a \geq 0$.
2. g is monotone decreasing: if $a_1 > a_2$ then $g(a_1) \leq g(a_2)$

We list some examples of popular decay functions:

Sliding Window. The function $g(a) = 1$ for $a < W$ and $g(a) = 0$ for $a \geq W$ captures the popular sliding window semantics that only considers items whose age is less than W. The parameter W is called the "window size".

Exponential Decay. The class of functions $g(a) = \exp(-\lambda a)$ for $\lambda > 0$ has been used for many applications in the past. Part of its popularity stems from the ease with which it

[1] For example, for a sliding window of size W, we can use our algorithms to find a time t such that the number of items arriving between t' and t is (approximately) W, and so reduce to the first version.

can be computed for many aggregates. It ensures that the time for g to drop by a constant fraction is the same, i.e. $g(a)/g(A+a)$ for a fixed A is the same for all a.

Polynomial Decay. For some applications, exponential decay is too fast, and a slower decay is required [9]. Polynomial decay is defined by $g(a) = (a+1)^{-\alpha}$, for some $\alpha > 0$ (Note the use of $(a+1)$ to ensure $g(0) = 1$). Equivalently, we can write $g(a) = \exp(-\alpha \ln(a+1))$.

Many other classes of decay functions are possible including super-exponential decays (e.g. $g(a) = \exp(-\lambda a^2)$) and sub-polynomial decays (e.g. $g(a) = (1+\ln(1+a))^{-1}$). Such functions are typically too fast or too slow for useful applications, so we do not consider them further, although they fit into our general framework. It is easy to verify that all the above functions satisfy both requirements for decay functions.

2.3 Time-decayed aggregates

We discuss techniques for coping with time-decayed aggregates. To sharpen the focus, we base our study on a few popular aggregates, range queries, quantiles and heavy hitters, since these are central to many stream computations. They are well-understood in the non-decayed case, but less well-studied under arbitrary decay functions, or on out-of-order streams.Our methods apply more broadly than to just these two aggregates, as discussed in Section 7. The definitions of time-decayed aggregates introduced below are implicit in some prior work, but have not previously been stated explicitly. In most cases, the definitions of time-decayed aggregates are natural and straightforward extensions of their undecayed versions.

Definition 2. *Given an input stream $S = \{\langle x_i, t_i\rangle\}$, the decayed weight of each item at time t is $g(a_i) = g(t - t_i)$. The* decayed count *of the stream at time t is $D(t) = \sum_i g(a_i)$ (we will refer to this as D when t is implicit).*

Our methods will, as a side effect, need to approximate D.

Definition 3. *The* decayed ϕ-quantile *of the stream is the item q satisfying $\sum_{i,x_i<q} g(a_i) \leq \phi D$, and $\sum_{i,x_i\leq q} g(a_i) > \phi D$. The* decayed ϕ-heavy hitters *are the set of items $\{p\}$ satisfying $\sum_{i,x_i=p} g(a_i) \geq \phi D$.*

One can easily verify that for $g(a) = 1$ for all a (no decay), these definitions equate to the standard definitions of quantiles and heavy hitters. With no decay, the timestamp t_i does not affect the result, and so there is little problem with out-of-order arrivals: the answer is independent of the input order. Thus, it is the presence of time decay which provides the challenge in handling out-of-order arrivals.

We assume that each item x_i is an integer in the range $[1 \ldots U]$. It is easy to verify that exact solutions to any of the above problems requires too much space. For example, exactly tracking the decayed count D in the sliding window decay model, requires space linear in the number of items within the sliding window [14]. It is well known that computing the quantiles exactly even without decay requires space linear in the input size; the same is true for exact tracking of heavy hitters. Therefore, we will need to tolerate some approximation in the answers in order to make the resource requirements manageable. We consider the following approximate versions.

Definition 4. *For* $0 < \epsilon < \phi \leq 1$*, the* ϵ-approximate decayed ϕ-quantiles problem *is to find an item* q *satisfying*
$(\phi - \epsilon)D \leq \sum_{i,x_i<q} g(a_i) \leq (\phi + \epsilon)D$.

For $0 < \epsilon < \phi \leq 1$*, the* ϵ-approximate decayed ϕ-heavy hitters problem *is to find a set of items* $\{p\}$ *satisfying*
$\sum_{i,x_i=p} g(a_i) \geq (\phi - \epsilon)D$*, and omitting no* q *such that*
$\sum_{i,x_i=q} g(a_i) \geq (\phi + \epsilon)D$.

Note that these problems can pose significant challenges. In particular, the answers depend significantly on the time at which the query is posed.

Example. Consider the input $\langle x, 3\rangle, \langle y, 2\rangle, \langle y, 1\rangle$ indicating that an item x arrived at time 3, while copies of item y arrived at times 2 and 1. If a decayed heavy hitters query with $\phi = \frac{1}{2}$ and a polynomial decay function $g(a) = (1 + a)^{-1}$ is posed at time 3, only x should be returned, since x has decayed weight 1 while y has decayed weight $\frac{1}{2} + \frac{1}{3} = \frac{5}{6}$, with threshold $\phi \cdot (1 + \frac{5}{6}) = \frac{11}{12}$. But if the same query is posed at time 4, then y should be returned, since x has decayed weight $\frac{1}{2}$, while y has decayed weight $\frac{1}{3} + \frac{1}{4} = \frac{7}{12}$, with threshold $\phi \cdot (\frac{5}{6} + \frac{7}{12}) = \frac{13}{24}$ □

Thus, even *a priori* knowledge of the decay function and query to be posed do not mean we can precompute and store a single result; instead we need to retain sufficient information to answer the query whenever it is posed.

Semantics of Late Arrivals. When out-of-order tuples arrive, we do not correct for any previous aggregate computations that would have been incorrectly reported. Instead, our output model ensures that these late arrivals are accounted for in any future aggregate computations. At query time, the aim is to approximate the answer that would be returned if all tuples returned so far had been returned exactly; this most honestly reflects the observer's knowledge. Note that simple solutions, such as trying to drop old tuples which do not contribute much to the answer does not in general give guaranteed space savings.

3 Sliding Window

In this section we show a general deterministic summary which allows us to answer queries for both quantiles and heavy hitters under sliding windows, where updates may arrive out-of-order.

3.1 Approximate Window Count

We first introduce a data structure which solves the simpler problem of tracking the decayed count of items within a sliding window as they arrive in an arbitrary order. This question has been studied in prior work explicitly [8, 7] and implicitly [11]. Our solution here matches the best space bounds for this problem [11, 7], and does so using a relatively simple construction which we can subsequently extend to answer other aggregates.

Given window size w (specified at query time) and a stream $\langle x_i, t_i \rangle$, we aim to approximate $D_w(t) = |\{i | t - t_i < w\}|$ with ϵ relative error. In our analysis, we assume that each t_i is an integer in the range $[0 \ldots W-1]$, and analyze the the space complexity as a function of W. Equivalently, W is an upper bound on the window size w. For simplicity, we assume that W is a power of 2—this does not lose any generality since W only has to be an upper bound on w. Our solution makes use of the q-digest data structure due to Shrivastava *et al.* [27]. We give a brief summary of our instantiation of the structure:

q-digest. Given a parameter $0 < \epsilon < 1$, the q-digest summarizes the frequency distribution f_i of a multiset defined by a stream of N items drawn from the domain $[0 \ldots W-1]$. The q-digest can be used to estimate the *rank* of an item q, which is defined as the number of items dominated by q, i.e. that is $\mathrm{r}(q) = \sum_{i<q} f_i$. The data structure maintains an appropriately defined set of *dyadic ranges* $\subseteq [0 \ldots W-1]$ and their associated counts. A *dyadic range* is a range of the form $[i2^j \ldots (i+1)2^j - 1]$ for non-negative integers i, j; i.e. its length is a power of two and it begins at a multiple of its length. It is easy to see that an arbitrary range of integers $[a \ldots b]$ can be uniquely partitioned into at most $2\log(b-a)$ dyadic ranges, with at most 2 dyadic ranges of each length. The q-digest has the following properties:

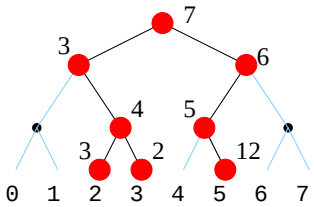
- Each range, count pair $(r, c(r))$ has $c(r) \leq \frac{\epsilon N}{\log_2 W}$, unless r represents a single item.
- Given a range r, denote its parent range as $\mathrm{par}(r)$, and its left and right child ranges as $\mathrm{left}(r)$ and $\mathrm{right}(r)$ respectively. For every $(r, c(r))$ pair, we have that $c(\mathrm{par}(r)) + c(\mathrm{left}(\mathrm{par}(r))) + c(\mathrm{right}(\mathrm{par}(r))) \geq \frac{\epsilon N}{\log_2 W}$.
- If the range r is present in the data structure, then the range $\mathrm{par}(r)$ is also present in the data structure.

Given query point $q \in [0 \ldots W-1]$, we can compute an estimate the rank of q, denoted by $\hat{r}(q)$, as the sum of the counts of all ranges to the left of q, i.e. $\hat{r}(q) = \sum_{(r=[l,h], c(r)), h<q} c(r)$, The following accuracy guarantee can be shown for the estimate of the rank: $\hat{r}(q) \leq \mathrm{r}(q) \leq \hat{r}(q) + \epsilon N$. Similarly, given a query point q one can estimate f_q, the frequency of item q as $\hat{f}_q = \hat{r}(q+1) - \hat{r}(q)$, with the following accuracy guarantee: $\hat{f}_q - \epsilon N \leq f_q \leq \hat{f}_q + \epsilon N$.

It is shown in [27, 11] that the q-digest can be maintained in space $O(\frac{\log W}{\epsilon})$. Updates to a q-digest can be performed in time $O(\log \log W)$, by binary searching the $O(\log W)$ dyadic ranges containing the new item to find the appropriate place to record its count. An example q-digest is shown in Figure 1(a). □

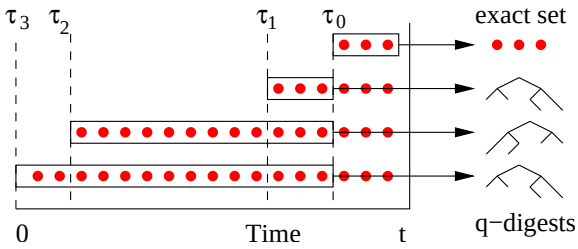
Sliding Window Count Algorithm. We solve the sliding window count problem using multiple instances of the q-digest data structure. Let the "right rank" of a timestamp τ, denoted by $\mathrm{rr}(\tau)$ be defined as the number of input elements whose timestamps are greater than τ. Given a window size $w \leq W$ at query time, the goal is to estimate $\mathrm{rr}(t-w)$ with relative error ϵ.

Theorem 1. *There is a data structure to approximate the sliding window count $D_w(t)$ with relative error no more than ϵ using space $O(\frac{\log W}{\epsilon} \log(\frac{\epsilon N}{\log W}))$. The time taken to update the*



q-digest with $(r, c(r))$ values of:
$([0,7],7),([0,3],3),([4,7],6),([2,3],4),$
$([4,5],5),(2,3),(3,2),(5,12).$

(a) q-digest with $W = 8$, $\epsilon = \frac{1}{2}$ and $N = 42$, so $\frac{\epsilon N}{\log W} = 7$.



(b) Data structure for sliding window range queries

Figure 1: Data Structure Illustrations

data structure upon the arrival of a new element is $O(\log(\frac{\epsilon N}{\log W}) \log\log W)$*, and a query for the count can be answered in time* $O(\log\log(\epsilon N) + \frac{\log W}{\epsilon})$.

Proof. Let $\alpha = \frac{3}{\epsilon}\log W$. Our algorithm maintains many data structures $Q_0, Q_1, \ldots$ (Figure 1(b)). Data structure Q_0 simply buffers the α elements with the most recent timestamps (ties broken arbitrarily). For $j > 0$, Q_j is a q-digest that summarizes recent elements of the stream. As j increases, the number of elements summarized by Q_j increases, but the error of estimates provided by Q_j also increases. The intuition is that if $\mathrm{rr}(t-w)$ is large, then to estimate $\mathrm{rr}(t-w)$ it suffices to use Q_j where j is large—the relative error is still small. On the other hand, if $\mathrm{rr}(t-w)$ is small, then a q-digest Q_j for small j suffices to answer the query, and the relative error is controlled.

Our use of the q-digest differs from the way it is employed in previous work [27] in the following way: while in [27], the upper bound on the count of a node in the q-digest (called the "count threshold" henceforth) increases with the number of elements being summarized, we fix the count threshold of a node in Q_j. The count threshold of Q_j is set to 2^j, and the maximum number of ranges within Q_j is bounded by α. Clearly, as more elements are added into Q_j, the number of ranges in Q_j will increase beyond α and we will need to discard some ranges. Informally, we retain the α "most recent" ranges in Q_j and discard the rest. More formally, the ranges within Q_j are sorted according to increasing order of right endpoints; ties are broken by putting smaller ranges first (as noted in [27], this corresponds to a post-order traversal of the tree implicitly represented by the q-digest). The α rightmost elements

Algorithm 3.1: SWUPDATE(t)

Input: arrival time t
$B \leftarrow B \cup \{t\}$
if $|B| > \frac{3 \log W}{\epsilon}$
then {
  $T \leftarrow \min t \in B$
  delete T from B
  for $j = 1$ **to** $\log(\frac{\epsilon N}{\log W})$
    do if $T > \tau_j$
    then QDINSERT(Q_j, T)

Algorithm 3.2: SWCOMPRESS()

for $j \leftarrow 1$ **to** $\log(\frac{\epsilon N}{\log W})$
do {
  recompute τ_j
  for each $(r, c(r)) \in Q_j$
    do if $\max(r) \leq \tau_j$
    then delete $(r, c(r))$ from Q_j
  QDCOMPRESS(Q_j)

Algorithm 3.3: SWQUERY(w)

Input: window size w
if $t - w \geq \tau_0$
  then return $(|\{\tau \in B | \tau > t - w\}|)$
  else {
    $\ell = \arg\min_i (\tau_i \leq t - w)$
    return $(\hat{\text{rr}}_\ell(t - w))$

Figure 2: Pseudocode for Sliding Window Count

in this sorted order are stored in Q_j.

For each q-digest $Q_j, j > 0$, we track τ_j, the minimum time such that all elements with timestamps greater than τ_j are properly summarized by Q_j. More precisely, τ_j is initialized to -1; anytime a range $[l, h]$ is discarded from Q_j, τ_j is set to $\max\{\tau_j, h\}$. Also, τ_0 is defined to be the greatest timestamp of an element discarded from Q_0, and -1 if Q_0 has not discarded any element so far.

For any $\tau \geq \tau_j$, $\mathrm{rr}(\tau)$ can be estimated using Q_j as the sum of the counts in all ranges $[l, h]$ in Q_j such that $l > \tau$. Suppose we denote this estimate by $\hat{\mathrm{rr}}_j(\tau)$, i.e. $\hat{\mathrm{rr}}_j(\tau) = \sum_{(r=[l,h],c(r)),l>\tau} c(r)$. The error in the estimate can only arise through ranges r in Q_j that contain τ (i.e. r neither falls completely to the left or completely to the right of r in Q_j). Since there are at most $\log W$ ranges that contain τ, the error in estimation is no more than $2^j \log W$. Thus we have the following if $\tau \geq \tau_j$:

$$\mathrm{rr}(\tau) \leq \hat{\mathrm{rr}}(\tau) \leq \mathrm{rr}(\tau) + 2^j \log W \tag{1}$$

It also follows that if Q_j is "full", i.e. the number of ranges in Q_j is the maximum possible, then $\mathrm{rr}(\tau_j) \geq \frac{3}{\epsilon} 2^j \log W - 2^j \log W$. Using the fact $\epsilon < 1$, we get:

$$\mathrm{rr}(\tau_j) > \frac{2^{j+1}}{\epsilon} \log W \tag{2}$$

Given window size w, our algorithm estimates $\mathrm{rr}\,(t-w)$ as follows. Let $\ell \geq 0$ be the smallest integer such that $\tau_\ell \leq t - w$. The algorithm returns $\hat{\mathrm{rr}}_\ell(t-w)$. The accuracy guarantee can be shown as follows. If $\ell = 0$, then the algorithm has returned the exact answer. Otherwise, we have from (1) that

$$0 \leq \hat{\mathrm{rr}}_\ell(t-w) - \mathrm{rr}(t-w) \leq 2^\ell \log W.$$

Also, since $\tau_{\ell-1} \geq t - w$, and $Q_{\ell-1}$ must be "full" (since otherwise $\tau_{\ell-1}$ would be -1) we have from (2) that

$$\mathrm{rr}(t-w) \geq \mathrm{rr}(\tau_{\ell-1}) > \tfrac{2^\ell}{\epsilon} \log W.$$

Thus the relative error $\frac{|\hat{\mathrm{rr}}_\ell(t-w) - \mathrm{rr}(t-w)|}{\mathrm{rr}(t-w)}$ is bounded by ϵ. The algorithm for updating the data structure, compressing it and for answering a query are given in Figure 2. The total space required depends on the total number of q-digests used. Due to the doubling of the count threshold each level, the largest q-digest that we need is Q_J for J given by $\frac{2^J}{\epsilon} \log W \geq N$, yielding $J = \lceil \log(\epsilon N) - \log\log W \rceil$. Thus the total space complexity is $O(\frac{\log W}{\epsilon} \log(\frac{\epsilon N}{\log W}))$. Each new arrival requires updating in the worst case all J q-digests, each of which takes time $O(\log\log W)$, giving a worst case time bound of $O(\log(\frac{\epsilon N}{\log W}) \log\log W)$ for the update. The query time is the time required to find the right Q_ℓ, which can be done in time $O(\log J) = O(\log\log(\epsilon N))$ (through a binary search on the τ_js) followed by summing the counts in the appropriate buckets of Q_ℓ, which can be done in time $O(\frac{\log W}{\epsilon})$, for a total query time complexity of $O(\log\log(\epsilon N) + \frac{\log W}{\epsilon})$. Each time the Compress routine is called, it takes time linear in the size of the data structure. Therefore, by running compress after every $O(\frac{\log W}{\epsilon})$ updates, the amortized cost of the compress is $O(\log(\frac{\epsilon N}{\log W})$, while the space bounds are as stated above. □

3.2 Range Queries, Quantiles, Heavy Hitters

To extend this approach to aggregates such as quantiles and heavy hitters, we use the same general structure as the algorithm for the count, but instead of just keeping counts within each q-digest, we will keep more details on each time range. We proceed in two steps, first by solving a particular range query problem, and then reducing our aggregates of interest to such range queries.

Definition 5. *A* sliding window range query *is defined as follows. Consider a stream of* $\langle x_i, t_i \rangle$ *tuples, and let* $r(w, x) = |\{i | x_i \leq x, t - t_i \leq w\}|$. *The approximate sliding window range query problem, given* (w, x) *with* $0 \leq w < W, 0 \leq x < U$, *is to return an estimate* $\hat{r}(w, x)$ *such that* $|\hat{r}(w, x) - r(w, x)| \leq \epsilon D_w(t)$.

The above setup is crucial for our subsequent reductions. Observe that the required approximation quality depends on $D_w(t)$, but not on the number of elements that dominate on the x coordinate. This corresponds to the requirements in Definition 4 for approximate quantiles and heavy hitters. The approximation guarantee required here is stronger than that required in prior work on range queries in data streams [17], so we need a modified approach to obtain it. Our algorithm for range queries combines the structure for approximate sliding window counts with an extra layer of data structures for range queries. The algorithm maintains many q-digests $Q_0, Q_1, \ldots$, each of which orders data along the "time" dimension—we call these the "time-wise" q-digests. Within $Q_j, j > 0$ the count threshold for each range is set to 2^{j-1}. Within each range $r \in Q_j$, instead of just keeping a count of the number of elements, we keep another q-digest, this time summarizing data along the value-dimension (similar to algorithms in [17]) —we call these the "value-wise" q-digests. We outline two alternate approaches which achieve slightly different bounds.

Eager merge version. The value-wise q-digests within Q_j are maintained based on a count threshold of $\frac{2^{j-1}}{\log U}$. Each value-wise q-digest for a timestamp range r summarizes the value distribution of all tuples whose timestamps fall within r—note that since the timestamp ranges within Q_j may overlap, a single element may be present in multiple (up to $\log W$) value-wise q-digests within Q_j. Similar to the count algorithm, Q_j also maintains a threshold τ_j, which is updated exactly as in the count algorithm.

To estimate $r(w, x)$, our algorithm uses Q_ℓ where ℓ is the smallest integer such that $\tau_\ell \leq t - w$. Within Q_ℓ, we find at most $\log W$ value-wise q-digests to query based on a dyadic decomposition of the range $(t - w, t]$, and query each of these for the rank of x. Finally, our estimate $\hat{r}(w, x)$ is the sum of these results. The error of the estimate has two components. Firstly, within the time-wise q-digest Q_ℓ we incur an error of up to $2^{\ell-1} \log W$ since we may undercount the number of elements within the timestamp range by up to $2^{\ell-1} \log W$. Next, within each value-wise q-digest, we incur an error of up to $\frac{2^{\ell-1}}{\log U} \log U = 2^{\ell-1}$. Since as many as $\log W$ value-wise q-digests maybe used, the total error due to the value-wise q-digests is bounded by $2^{\ell-1} \log W$. Hence the total error in the estimate is bounded by $2 \cdot 2^{\ell-1} \log W = 2^\ell \log W$. By choosing $\alpha = \frac{3}{\epsilon} \log W$ ranges within each each Q_j, and using a

similar argument to the count algorithm, we get $D_w \geq \text{rr}(\tau_{\ell-1}) > \frac{2^\ell \log W}{\epsilon}$. Thus the error in our estimate of $r(w, x)$ is no more than ϵD_w, as required.

Space and Time Complexity. Note that the sum of counts of all nodes within all value-wise q-digests within Q_j is $O(\log W \, \text{rr}(\tau_j)) = O(\frac{1}{\epsilon} 2^j \log^2 W)$, since each element maybe included in no more than $\log W$ value-wise q-digests within Q_j. Consider any triple of (parent, left child, right child) ranges within a value-wise q-digest. The total count of these triples must be at least $\frac{2^{j-1}}{\log U}$, implying that for this many counts, a constant amount space is used. Thus, the total space taken to store Q_j is $O(\log^2 W \log U/\epsilon)$. As analyzed before, there are $O(\log(\frac{\epsilon N}{\log W}))$ different timewise q-digests, leading to a total space complexity of $O(\frac{1}{\epsilon} \log(\epsilon N/\log W) \log^2 W \log U)$. Consider the time to update each Q_j: This requires the insertion of the element into no more than $\log W$ value-wise q-digests; each such insertion takes time $O(\log \log U)$ and hence the total time to insert into all Q_js is $O(\log(\frac{\epsilon N}{\log W}) \log W (\log \log U))$
$= O(\log(\epsilon N) \log W (\log \log U))$. □

Defer Merge version. In the defer merge version, we use a similar idea, of time-wise q-digests Q_j, each node of which contains a value-wise q-digest. Here, we have the same number and arrangement of time-wise q-digests, but modify how we use the value-wise structures. Instead of inserting each update in all value-wise q-digests that summarize time ranges in which it falls, we insert it in only one, corresponding to the node in the time-wise structure whose count we increment due to insertion. The pruning condition for the value-wise q-digest is based on $\epsilon n/2 \log U$, where $n = c(r)$ is the number of items counted by the time-wise q-digest in the range. In other words, each value-wise q-digest is just a "standard" q-digest which summarizes the values inserted into it, and so takes space $O(\frac{\log U}{\epsilon})$.

To answer queries $r(w, q)$, we again find τ_ℓ based on w and query Q_ℓ. Again, we incur error $2^{\ell-1} \log W$ from uncertainty in Q_ℓ. We then merge together all value-wise summaries within Q_ℓ which correspond to items arriving within the time window $(t-w, t]$ (note that we have deferred this merging to query time, as opposed to the above "eager merge" approach, which computes the result of this merging at insertion time). We pose the query x to the resulting q-digest. By the properties of merging q-digests, the error in this query is bounded by $\frac{\epsilon}{2} D_w$. Summing these two components gives the required total error bound of ϵD_w. **Space and Time Complexity.** We compute the space required by taking the number of value-wise q-digests for each Q_j, $O(\frac{\log W}{\epsilon})$ and multiplying by the size of each, $O(\frac{\log U}{\epsilon})$, over the $J = \log \epsilon N - \log \log W$ levels. The overall bound is $O(\frac{1}{\epsilon^2} \log U \log W \log(\frac{\epsilon N}{\log W}))$. (thus we trade off a factor of $\log W$ for one of $\frac{1}{\epsilon}$ compared to the above eager-merge version). To perform an insertion, for each Q_j we perform an insertion into the time-wise q-digest, and then into the value-wise q-digest, in time $O(\log \log U + \log \log W)$. The amortized cost of compressing can be made $O(1)$ by the same argument as above. The overall amortized cost per update is therefore $O(\log(\frac{\epsilon N}{\log W})(\log \log W + \log \log U))$. □

Summarizing these two variant approaches, we conclude:

Theorem 2. *Sliding window range queries can be approximated in space*

$O(\frac{1}{\epsilon}\log U \log W \log(\frac{\epsilon N}{\log W})\min(\log W, \frac{1}{\epsilon}))$ *and time* $O(\log(\frac{\epsilon N}{\log W})\log W \log\log U)$ *per update. Queries take time linear in the space used.*

Reductions of Quantiles and Heavy Hitters to Range Queries. We now show that answering heavy hitters and quantiles queries in a sliding window can be reduced to range queries, and that approximate answers to range queries yield good approximations for quantiles and heavy hitters. For a maximum window size W, we create the data structure for range queries with accuracy parameter $\frac{\epsilon}{2}$. To answer a query for the approximate ϕ-quantile we first compute an approximation $\hat{D}_w$ of D_w using the time-wise q-digests, through the results of Theorem 1. We then binary search for the smallest x such that $\hat{r}(w,x) \geq \phi\hat{D}_w$. Observe that such an x satisfies the requirements for being an approximate ϕ-quantile: we have $|\hat{D}_w - D_w| \leq \frac{\epsilon}{2}D_w$, and $|\hat{r}(w,x) - r(w,x)| \leq \frac{\epsilon}{2}D_w$. Combining these gives the required bounds from Definition 4.

One way to answer ϕ-heavy hitter queries is to observe that we can pose quantiles queries for ϕ'-quantiles, for $\phi' = \epsilon, 2\epsilon, 3\epsilon \ldots 1$. All items that repeatedly occur as $\frac{\phi}{\epsilon}$ (or more) consecutive quantiles are reported. Note that if any item has frequency at least $(\phi+\epsilon)D_w$, it will surely be reported. Also, any item which has frequency less than $(\phi-\epsilon)D_w$ will surely not be reported.

Corollary 1. *We can answer approximate sliding window quantile and heavy hitters with out-of-order arrivals in the same bounds as Theorem 2.*

This lets window size $w<W$ to be specified at query time; if instead we fix it to W tuples and keep only the q-digest for the appropriate τ_j, we save a factor of $O(\log(\frac{\epsilon N}{\log W}))$.

4 Exponential Decay

Exponential decay covers the class of decay functions $g(a) = \exp(-\lambda a)$ for a constant $\lambda > 0$. We can compute a summary for computing aggregates under such a decay function which requires no more space than the summary with no decay (i.e. $g(a) = 1$). These techniques rely explicitly on the fact that, for any exponential decay function, $g(a + A)/g(a) = g(A)$ for all a and A.

4.1 Quantiles

We consider the q-digest summary structure described above, which can be used to answer uniform quantile queries. Observe that: (1) The q-digest does not require that all items have unit weight, but can be modified to accept updates with arbitrary (i.e. fractional) non-negative weights. (2) Multiplying all counts in the data structure by a constant γ gives an accurate summary of the input scaled by γ. It is easy to check that the properties of the data structure still hold after these transformations, e.g. that the sum of the counts is D, the sum of the (possibly scaled) input weights; no count for a range exceeds $\frac{\epsilon D}{\log U}$ etc.

Thus given an item arrival of $\langle x_i, t_i \rangle$ at time t, we can create a summary of the exponentially decayed data. Let t' be the last time the data structure was updated; we multiply every count in the data structure by the scalar $\exp(-\lambda(t-t'))$ so that it reflects the current decayed weights of all items, and then update the q-digest with the item x_i with weight $\exp(-\lambda(t-t_i))$. Note that this may be time consuming, since it affects every entry in the data structure; we can be more "lazy" by tracking D, the current decayed count exactly[2], and keeping a timestamp t_r on each counter $c(r)$ denoting the last time it was touched. Whenever we require the current value of range r, we can multiply it by $\exp(-\lambda(t-t_r))$, and update t_r to t. This ensures that the asymptotic space and time costs of maintaining an exponentially decayed q-digest are as before. To see the correctness of this approach, let $S(r)$ denote the subset of input items which the algorithm is representing by the range r: when the algorithm processes a new update $\langle x_i, t_i \rangle$ and updates a range r, we (notionally) set $S(r) = S(r) \cup i$; when the algorithm merges a range r' together into range r by adding the count of (the child range) r' into the count of r (the parent), we set $S(r) = S(r) \cup S(r')$, and $S(r') = \emptyset$ (since r' has given up its contents). We argue that our algorithm maintains the property that $c(r) = \sum_{i \in S(r)} g(a_i)$. It is easy to check that every operation which modifies the counts (adding a new item, merging two range counts, applying the decay functions) maintains this invariant. In line with the original q-digest algorithm, every item summarized in $S(r)$ is a member of the range r, i.e. $i \in S(r) \Rightarrow x_i \in r$, and at any time each tuple i from the input is represented in exactly one range r.

To estimate $\mathrm{r}_\lambda(x) = \sum_{i, x_i \leq x} g(a_i)$, we compute

$$\hat{r} = \sum_{r=[l \ldots h], h \leq x} c(r).$$

By the above analysis of $c(r)$, we correctly include all items that are surely less than x, and omit all items that are surely greater than x. The uncertainty depends only on the ranges containing x, and the sum of these ranges is at most $\epsilon \sum_r c(r) = \epsilon D$. Thus, we answer any decayed rank query deterministically, with bounded approximation error. For quantile queries, we can quickly find a ϕ-quantile with the desired error bounds by binary searching for x whose approximate rank is ϕD. We can also finds ϵ-approximate exponentially decayed heavy hitters in the same bounds. In summary,

Theorem 3. *Under a fixed exponential decay function $g(a) = \exp(-\lambda a)$, we can answer approximate decayed quantile and heavy hitter queries in space $O(\frac{1}{\epsilon} \log U)$ and time per update $O(\log \log U)$. Queries take time $O(\frac{\log U}{\epsilon})$.*

4.2 Heavy Hitters

Prior work by Manjhi *et al.* [20] has shown that computing Heavy Hitters under exponential decay is possible by modification of algorithms for the problem without decay. We take a similar tack, but our approach means that we can also easily accomodate out-of-order arrivals, which is not the case in [20].

[2]Note that under exponential decay it is easy to track D exactly using a single counter and timestamp, in contrast to sliding window and polynomial decay, which require more sophisticated algorithms to approximate D in small space.

Algorithm 4.1: HEAVYHITTERUPDATE(x, t_x, t)

Input: item x, arrival time t_x, current time t
if $\exists i.\, \text{item}[i] = x$
  then $i \leftarrow \text{item}^{-1}(x)$
  else $\begin{cases} i \leftarrow \arg\min_j(\text{count}[j]) \\ \text{item}[i] \leftarrow x \end{cases}$
$\text{count}[i] \leftarrow \text{count}[i] \cdot (g(t) - g(\text{time}[i])) + g(t - t_x)$
$\text{time}[i] \leftarrow t$

Figure 3: Pseudocode for Heavy Hitters with exponential decay

A first observation is that we can use the same (exponentially decayed) q-digest data structure to also answer heavy hitters queries, since the data structure guarantees error at most ϵD in the count of any single item; it is straightforward to scan the data structure to find and estimate all possible heavy hitters in time linear in the data structure's size. Thus Theorem 3 also applies to heavy hitters. However, we can reduce the space required somewhat, and extend to the case when the input is drawn from an arbitrary domain. We outline the approach since it is similar in spirit to the above analysis for quantiles.

The data structure of [22] gives a way to track heavy hitters over data streams using $O(\frac{1}{\epsilon})$ counters and item names. As above, one can work through the definition of the data structure and verify that it can be modified to accept arbitrary non-negative weighted updates, and multiplied through by a scalar to effectively summarize the input multplied by this scalar. Thus the same reduction works to keep an exponentially decayed data structure: whenever an item arrives, we insert the item with its decayed weight into the data structure, and ensure that all other counts are decayed to the correct extent (either on every arrival, or lazily, as in Figure 3). To answer heavy hitters queries, we simply pass over the stored items and their counts, and output all items whose approximated counts are above ϕD. One can show that the error in estimating any count is at most ϵD. Thus,

Theorem 4. *Under a fixed exponential decay function $g(a) = \exp(-\lambda a)$, we can answer approximate decayed heavy hitters in space $O(\frac{1}{\epsilon})$ and update time (expected) $O(1)$.*

5 Polynomial and General Decay

As explained in Section 2.2, a user may need a decay function other than sliding windows or exponential decay. We describe two methods to handle more general decay functions, including polynomial decay. The first method applies to *any* decay function by reducing a query for decayed sums/counts to multiple sliding windows queries (as in [9]). We describe how to execute this reduction efficiently (time-wise). The second method, "Value-Division",

is potentially more efficient, and applies to a broad class of "smooth" decay functions (defined formally below); this class includes polynomial decay. We focus the discussion on polynomial decay for ease of exposition.

5.1 Reduction to Multiple Sliding Windows

We first outline a generic approach to computing decayed aggregates by reduction to multiple sliding windows. This is based on the observations in [9] for computing decayed sums and counts. Consider an arbitrary decay function $g(a)$ and the heavy hitters aggregate. We can write the (decayed) count of any item x as the sum

$$g(0)f_x(0) + \sum_{j=1}^{t}(g(j) - g(j-1))f_x(j),$$

where $f_x(j)$ denotes the count of item x in the window of size j. Thus we can find the (approximate) heavy hitters if we find the (approximate) counts of each item in each window up to t. This shows that in the space bounds given by Theorem 2 we can apply arbitrary decay functions. Because the count of x in window j is approximated with error ϵD_j, summing all counts gives the required error:

$$\sum_{j=1}^{t} g(t-j) - g(t-j+1)\epsilon D_j = \epsilon D.$$

Note that this algorithm as stated is not time efficient. Yet it is straightforward to extend it to run more quickly, by making use of the contents of the data structure. Firstly, observe that we do not have to enumerate every possible item x; rather, we can use the information on which items are stored in the sliding window data structure, since items not stored are guaranteed not to be significant under the decay function. Secondly, we do not have to query all possible time values. Again, the data structure only stores information about a limited number of time stamps, and queries about sliding windows with other timestamps will give the same answers as queries on some timestamp stored in the data structure. Thus we have to evaluate the sum only at timestamps stored in the data structure, rather than all possible timestamps.

For quantiles, the results are similar. Instead of computing the decayed count of an item, we compute the decayed rank of items, and can binary search to find the desired quantile in the usual way. The same space bounds hold. Note that these are very strong results: it says that not only can we handle item arrivals in completely arbitrary orders, but also we can apply any arbitrary decay function efficiently, and this decay function can be specified at query time, after the input stream has been seen—and all these results hold deterministically. Combining these gives:

Theorem 5. *We can answer decayed heavy hitter and quantile queries on out-of-order arrivals in the bounds stated in Theorem 2. Queries take time linear in the space.*

5.2 Value-Division

The above algorithm is quite general, but we may be able to improve the space bounds and time cost by using an alternate approach. We generalize the technique of Cohen and

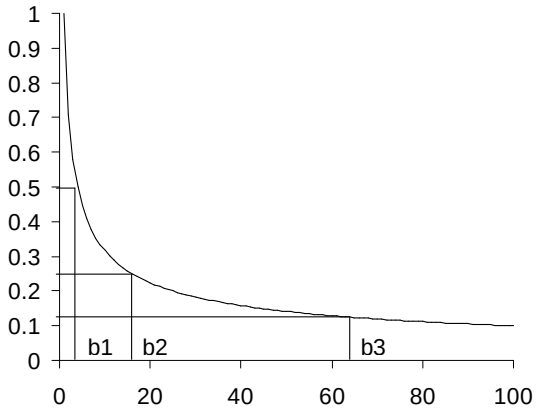


Figure 4: Choice of boundaries in value-division approach

Strauss for decayed sums and counts [9] to apply to quantiles and heavy hitters. In fact, they apply to a broad range of aggregates, comprising any aggregate which has a summary such that we can merge two summaries to get a summary of the union of the inputs, and scale a summary to get a summary of the linearly scaled input. This incorporates most "sketch" algorithms, as well as simple aggregates such as sums and counts. We refer to these as "linear summaries". The technique allows us to approximate decayed aggregate, based on tracking a set of "value divisions" or boundaries.

Smooth Decay Functions. If decay function g is continuous, then let $\dot{g}(x)$ denote the derivative of g at age x.

Definition 6. *A decay function g is defined to be* smooth *(or "smoothly decaying") if for all $a, A > 0$,*

$$\dot{g}(a)g(a+A) \leq \dot{g}(a+A)g(a).$$

Note that Sliding Window is not smooth, since it is not continuous. Exponential Decay is smooth (the definition holds with equality), as is Polynomial Decay.

Value divisions. Given a smooth decay function g, we define a set of boundaries on ages, b_i, so that $g(b_i) = (1+\theta)^{-i}$. We keep a small number of (linear) summaries of the input. Each summary s_j will correspond to items drawn from the input within a range of ages. These ranges fully partition the time from 0 to t, so no intervals overlap. Thus summary s_j summarizes all items with timestamps between times t_j and t_{j+1} (see Figure 4).

We use the boundaries to define the summary time intervals: we ensure that for all boundaries b_i at time t, there is at most one summary s_j such that

$$(t - b_i) < t_{j+1} < t_j < (t - b_{i+1}).$$

To maintain this, if we find a pair of adjacent summaries $j, j+1$ such that

$$(t - b_i) < t_{j+2} < t_j < (t - b_{i+1})$$

(i.e. both summaries fall between adjacent boundaries), then we merge summaries s_j and s_{j+1}, which now summarize the range t_j to t_{j+2}.

Note that the time ranges of the summaries, and the way in which they are merged, depends only on the time and the boundaries, and not on any features of the arrival stream. This naturally accommodates out of order arrivals (when a new arrival tuple $\langle x_i, t_i \rangle$ has a t_i value that precedes other t_j values already seen). Since the summaries partition the time domain, to process the update we just have to find the summary which covers t_i, and include the item in that summary. This works because the notion of time is independent of the arrivals.

Theorem 6. *Given a linear summary algorithm, we can build a $(1+\theta)$ accurate answer to (polynomial) decay queries by storing $O(\log_{1+\theta} g(t))$ summaries.*

Proof. We first show that we can build an accurate summary by combining stored summaries, and then show a bound on the number of summaries stored.

Observe that for any summary s_j whose age interval falls between two adjacent boundaries b_i and b_{i+1}, we have

$$b_i \leq t - t_{j+1} \leq t - t_j \leq b_{i+1},$$

and so

$$g(b_i) \geq g(t - t_{j+1}) \geq g(t - t_j) \geq g(b_{i+1}) = g(b_i)/(1+\theta)$$

(by the fact that g is monotone decreasing). The monotonicity of g also ensures that the g value of every item summarized by s_j is between $g(t - t_{j+1})$ and $g(t - t_j)$, which are within a $1+\theta$ factor of each other. Thus, we can proceed as if every item in s_j arrived at time t_j, and only affect the result by a factor of at most $(1+\theta)$.

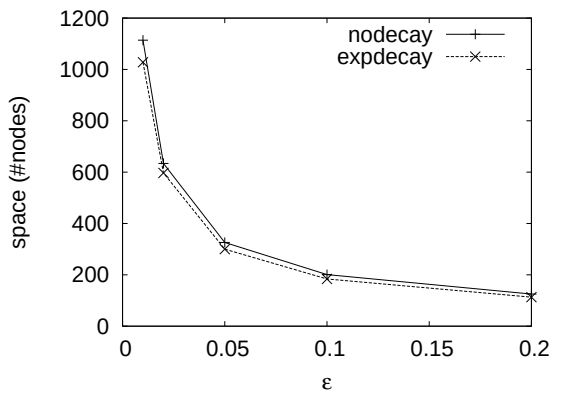
The same is true for any summary which straddles any boundary b_i (i.e. $(t - t_j) \geq b_i \geq (t - t_{j+1})$). We argue that at some earlier time, s_j fell between two adjacent boundaries: by induction, either this is true from when s_j is created as a summary of a single time instant; or else, s_j is formed as the merge of two summaries and the resultant summary fell between two boundaries. Either way, we can argue that, at the time of formation $g(t - t_j)/g(t - t_{j+1}) \leq (1+\beta)$. We now argue that if g is smoothly decaying, this remains true for all times $T > t$. Let $a = (t - t_j)$ and $A = (t_{j+1} - t_j)$, and analyze $\frac{d}{da} g(a)/g(a+A)$: this is non-increasing if

$$g(a+A)\dot{g}(a) - g(a)\dot{g}(a+A) \leq 0,$$

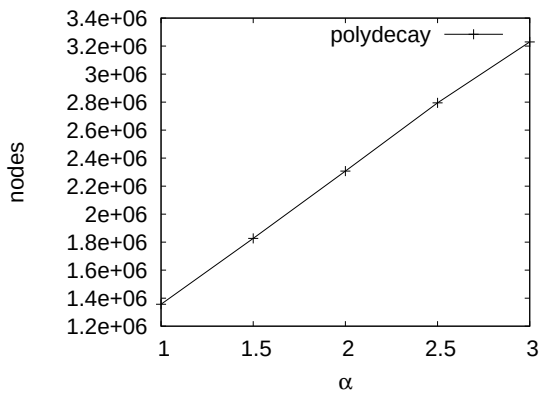
by standard differentiation and the chain rule. But this is precisely the condition that g is smoothly decaying, thus we ensure that for every summary, treating all items summarized as if they all arrived at the same time t_j only affects the result by a factor of at most $(1+\theta)$.

In order to answer queries, we use a similar idea to that in the general approach above. For each summary, we take the age of the most recent item summarized, a, and linearly scale the summary by $g(a)$, and merge all the scaled summaries together. We answer the query by probing this scaled and merged summary. Correctness follows by observing that since the range of ages of items in the summary is bounded by a $(1+\theta)$ factor, the error introduced by treating all items as the same age is at most this much.

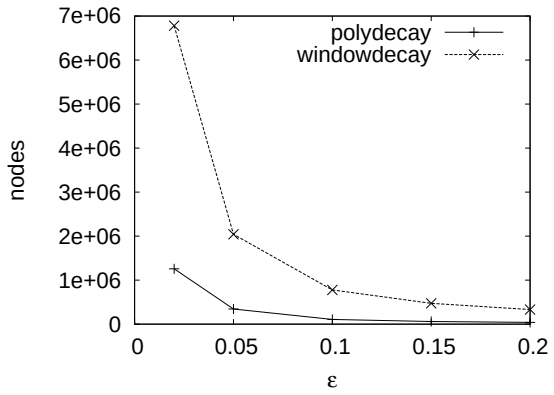
The number of summaries stored is bounded in terms of the duration of the data (or on a cut off point W beyond which we force $g(a > W) = 0$). At any instant, each summary either falls between two boundaries, or crosses a boundary. There is at most one summary



(a) exp ($\lambda = 0.1$) and no-decay: space vs ϵ



(b) poly decay: space vs α ($\epsilon = 0.1$)

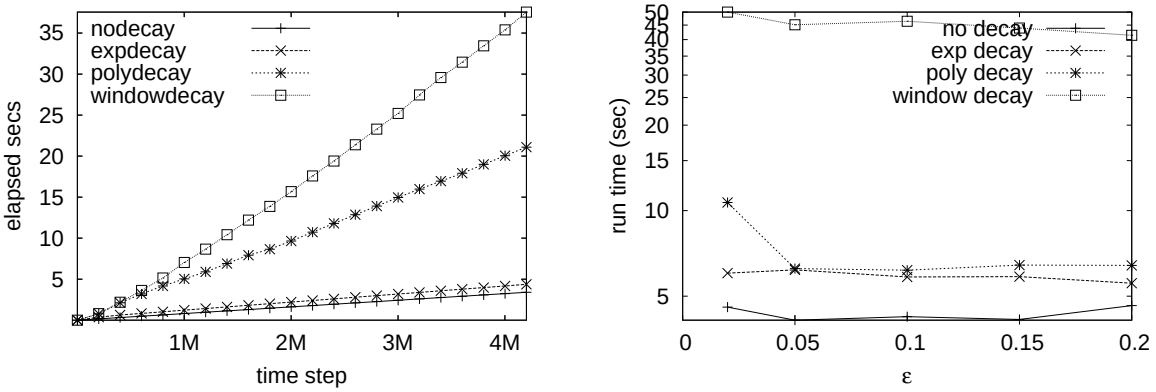


(c) poly ($\alpha = 2$) and window: space vs ϵ

Figure 5: Space usage of (a) exp vs no-decay; (b) poly decay wrt α; and (c) poly vs window decay, using flow data.

falling between each boundary, which we associate with the boundary to its left; therefore the number of summaries stored is equal to twice the number of boundaries which have input items older than them. The final such boundary, b_k, therefore satisfies $g(t) \geq b_k = (1+\theta)^{-k}$, since the oldest item has age at most t. Thus, $k = -\ln(g(t))/\ln(1+\theta)$, and hence the number of summaries is $O(\frac{1}{\theta}\ln(\frac{1}{g(t)}))$. □

Example: Decayed Quantiles with polynomial decay. We demonstrate the result when applied to computing time-decayed quantiles with polynomial decay, i.e. $g(a) = \exp(-\alpha \ln(1+a))$. Using the q-digest again, regular quantiles can be answered with error β using a summary of size $O(\frac{\log U}{\beta})$, where U denotes the size of the domain from which the quantiles are drawn. The data structure is a linear summary (as shown in the previous section), and so can be used with polynomial decay. The total space required is therefore $O(\frac{1}{\theta}\ln(1/g(t)) \cdot \frac{\log U}{\beta}) = O(\frac{\alpha}{\theta\beta}\ln t)$ for polynomial decay. The total approximation error is, in the worst case, $(\theta+\beta)D$. In order to guarantee overall error of ϵD, we ensure that $\theta+\beta = \epsilon$;



(a) performance vs timestamp (World Cup)

(b) performance vs ϵ (flow)

Figure 6: Performance of the various algorithms

the space is minimized by $\theta = \beta = \epsilon/2$, giving $O(\frac{\log U}{\epsilon^2} \log t)$ overall. The time cost will depend on the number of merges, which in turn depends on the distribution of timestamps.

Observe that this is comparable to the general bound we have for arbitrary decay functions by reduction to sliding window queries. Both algorithms have terms in $O(\frac{1}{\epsilon} \log U \log W)$, the value division approach has another $O(\frac{1}{\epsilon})$ factor whereas the sliding window approach has an $O(\min(\frac{1}{\epsilon}, \log W) \log \epsilon N)$ factor. So asymptotically the space bounds are better for value division in the case that $\frac{1}{\epsilon} \leq \log W$, and should be competitive for other values.

Extensibility: Decay Domination. We have so far assumed that the decay function is known *a priori*, since $g(a)$ is used to set the boundary values. However, observe that if we create boundaries based on $g(a) = (1+a)^{-2}$, we obtain boundaries at $a = 1, (1+\theta)^{1/2}, (1+\theta), (1+\theta)^{3/2} \ldots$. This is a superset of the boundaries we would have created for $g'(a) = (1+a)^{-1}$ (that is, $a = 1, (1+\theta), (1+\theta)^2) \ldots$), and so it can be shown that the data structure used for $g(a)$ can also be used for $g'(a)$. More strongly, we have:

Lemma 1. *Given the results of running the value-division algorithm with boundaries b_i based on decay function g and parameter θ, at query time, we can apply any smooth decay function g' and build a $(1+\theta')$ accurate answer, provided that $\forall i. g(b_i)/g(b_{i+1}) \leq (1+\theta')$.*

Thus, we can set the boundaries based on a particular function g and θ value, and be able to specify a new function g' that is "weaker" than g (decays less quickly), getting a guarantee with a θ' that is better than the original θ. Equally, we can specify a g' that is stronger than g (decays faster), and still obtain a result, but with larger θ': creating boundaries based on θ and $g(a) = (1+a)^{-\alpha}$ gives boundaries that are valid for $g'(a) = (1+a)^{-2\alpha}$ with $\theta' = 2\theta + \theta^2$. Note however that using boundaries intended for a polynomial decay function with an exponential decay function will give poor guarantees, since $\theta' = O(\theta \ln g'(t) / \ln g(t))$.

6 Experiments

While the asymptotic complexities derived in the preceding sections give an indication of the relative efficiency of the algorithms, in this section we evaluate the space usage and performance of them in practice. We compare exponential, polynomial and sliding window decay functions. As a baseline, we also present results using no decay function which does not incur the overhead from having to deal with out-of-order arrivals since non-decayed aggregates are not affected by arrival order.

6.1 Experimental Setup

We implemented our core methods based on q-digests for quantiles under a variety of decay functions, and measured the space usage and processing time to better understand their relative performance. For comparability, our methods used the same underlying implementations of q-digests, in C. We report space usage in terms of the number of nodes stored in the data structures since the nodes of the respective data structures all store tuples of roughly the same space: 12 bytes per node in our implementation. For the sliding window algorithms, we compared the eager merge and defer merge strategies. On our data sets the defer-merge approach was more efficient in both time and space, so we report these results only.

We show results on two different network data streams. The first consists of 5 million records of IP flow data aggregated at an ISP router using Cisco NetFlow [25], projected onto `(begin_time, num_octets)` but sorted on `end_time`; consequently there is moderate disorder on the arrivals. The second data set is 5 million records of Web log data collected during the 1998 Football World Cup [18], projected onto `(time, num_bytes)`. Additional out-of-order arrivals were introduced to it by including data from multiple subsequent days with the only the time of day used: consequently the timestamps "loop" several times. All experiments were run on a 2.8GHz Pentium Linux machine with 2 GB main memory.

6.2 Space Usage

We first compare the space usage of our algorithm for exponentially decayed aggregates against the (non-decayed) q-digest. As we showed in Section 4, these two have the same asymptotic bounds. Figure 5(a) graphs the space usage of the two decay functions for different values of ϵ on flow data. It shows that in practice there is very little space overhead for exponential decay, as expected. The curves for these decay functions using World Cup data looked almost identical and are omitted due to space constraints.

To get a sense of how the polynomial degree α affects space usage of polynomial decay in practice, we plot space at different values of α using the value division approach with ϵ fixed at 0.1 in Figure 5(b). As indicated by our asymptotic analysis, the relationship is close to linear. We chose a reasonable degree of $\alpha = 2$ and compared the value division approach with the sliding window approach (with $W = 2^{20}$), both of which can be used

to maintain streaming aggregates with polynomial decay. Figure 5(c) plots space against ϵ for these two approaches. It shows that the value division approach is significantly more space-efficient than sliding windows, especially with small values of ϵ (results using the flow data are shown).

Recall that for both data sets, the total input size is 5 million items. Thus, for moderate values of ϵ, the space used by our approximate algorithms (especially value-division) is significantly less than the size of the input, by up to an order of magnitude. As ϵ decreases, the space rises sharply, especially for the sliding window algorithms. In the worst case, the size exceeds that of the input, since each input item is represented in multiple time-wise q-digests Q_j. Meanwhile, the value-division approach always uses fewer nodes that the number of input tuples, since each input item is represented at most once in the structure.

6.3 Performance

Figure 6(a) compares the time (in seconds) taken to update the data structure for the exponential, polynomial and sliding window decay functions as well as the q-digest (no-decay) as a baseline, at increasing timestamps using World Cup data. With flow data (not shown), the curve ordering was the same but the smaller amount of disorder enabled the value division approach to achieve performance closer to that of exponential decay. Figure 6(b) shows how these times vary with ϵ, on a log scale. There is little variation in these times, in accordance with our analysis which shows a weak dependence on ϵ in the running times. A small exception is for value division on polynomial decay. In fact, this is more due to our non-optimal implementation: to insert a new item, the code searches through the time ranges of the summaries in order from the most recent. (For small ϵ, there are more summaries to search through.) Similarly, on highly out-of-order data, the running time increased due to more levels of the sliding window algorithms. An improved implementation with a dynamic index on the timestamps would improve the cost.

Overall, exponential decay can handle a throughput of around 1 million updates per second, value division around a quarter of this rate, while sliding window still achieves around 100 thousand items per second on our set up. It is highly effective to implement exponential decay function, since the overhead compared to no decay is small. The cost of other decay functions is higher; for polynomial decay, it seems generally better to use the value-division approach. Lastly, the extra flexibility and sophistication that the sliding windows approach provides can be obtained but at a premium (up to an order of magnitude here).

7 Extensions

7.1 Distributed Observations

We note that the algorithms proposed here all work in a distributed setting, to varying degrees. That is, if two separate observers see independent streams, we can merge their

summaries to make a sketch of the union of the streams. The exponential decay case is easiest to describe: since a fundamental property of the q-digest data structure is that multiple q-digests can be merged to give the same space and accuracy guarantees as a single digest, it is easy to check that this still holds in the exponentially decayed case, giving bounds for quantiles and heavy hitters.

The same idea applies in the sliding window case: since the data structures are formed from the combination of q-digests, by doing the appropriate merging and pruning, one can give the required guarantees. We omit the details here. Lastly, the value division approach also allows the data structures to be merged. Where we have two (possibly overlapping) summaries which fall between two adjacent boundaries, we can merge them into a single summary. But in the case where we have summaries which cross the same boundary, we cannot immediately merge them, since the resulting summary might span an excessively long time period. So we can answer queries correctly by keeping both summaries, but, as we merge the sketches of more and more streams together, we are not able to guarantee in the worst case that the resulting data structure will have bounded size less than the sum of the sizes of the input sketches, so this remains an open problem in this area.

7.2 Duplication and Deletions

In many of the motivating scenarios—the confluence of massive data streams—it is possible that in addition to out of order arrivals, we may also need to deal with other data quality issues. In particular, our methods described so far do not deal with duplicate arrivals. In some settings, we may (through retransmissions etc.) see the same data more than once; however, it should only be counted once. A standard approach within data streams is to replace the exact counter with "approximate count distinct": randomized sketches which approximate the number of unique items they have observed. This general idea is used in [13], using Flajolet-Martin sketches [15]. However, the space cost of replacing every single counter (integer) with approximate counters (each maybe kilobytes in size) is significant. An alternate approach for the sliding window counting problem with duplicates is given by [16], based on the technique of "distinct sampling".

Another case to study is that of deletions, or retractions: updates which cancel out a prior update. Such updates can occur in streams where an error is detected and must be corrected, or when quantities are being monitored which can decrease as well as increase. We observe that in certain cases these can be handled, by replacing the deterministic q-digests with randomized sketches, such as the Count-Min sketch [12]. For example, in the exponential decay case, we can simply keep a CM sketch where every entry is an exponentially decayed. A deletion of a prior update can be handled by computing the current decayed weight of the update, and subtracting it from the counters where it is stored. Similarly, using CM sketches in the value-division algorithm allow deletions to be processed. Note that this requires that the deletion specify the full item identifier, original weight, and timestamp of the update to delete.

It remains open to process deletions in the sliding window case for holistic aggregates. The

Reference	Aggregate	Space Bound	Decay	Out-of-order
[20]	Heavy Hitters	$O(1/\epsilon)$	Exponential	No
Here	Heavy Hitters	$O(1/\epsilon)$	Exponential	Yes
Here	Quantiles	$O(\log U/\epsilon)$	Exponential	Yes
[19]	Heavy Hitters	$O(\frac{1}{\epsilon}\log \epsilon N)$	Sliding Window	No
[4]	Heavy Hitters	$O(\frac{1}{\epsilon}\log^2 \frac{1}{\epsilon}\log \epsilon N)$	Sliding Window	No
[4]	Quantiles	$O(\frac{1}{\epsilon}\log \frac{1}{\epsilon}\log \epsilon N \log N)$	Sliding Window	No
Here	Quantiles/Heavy Hitters	$O(\frac{1}{\epsilon}\log U \log W \log \epsilon N \min(\log W, \frac{1}{\epsilon}))$	Sliding Window	Yes
Here	Quantiles	$O(\frac{1}{\epsilon^2}\log U \log W)$	Poly	Yes

Table 1: Comparison of properties of algorithms for decay functions on streams

case of both deletions and duplicates has also not been well-studied: note that the methods outlined above apply to only one of deletions and duplications, not both. In general little has been proven about algorithms over streams with both deletions and duplications, even outside of time decay.

7.3 System Issues

So far we have focused on two important holistic aggregates, quantiles and heavy hitters, along with rank queries and decayed sums. However, the methods we describe are quite general. Streaming data management systems (DSMSs) such as AT&T's Gigascope allow users to specify code to support arbitrary aggregates in the form of User Defined Aggregate Functions [10]. The UDAF author defines a small number of functions, which are called appropriately by the system when needed. A similar approach can work to support decayed aggregates (specified, e.g., in a `DECAY BY` clause), requiring some additional routines. For example, exponential decay can be supported by exporting a `Scale` routine, so that the DSMS can multiply the summary by a scalar when needed to perform the decay. For value division, we needed to be able to `Scale` and also `Merge` together two summaries. By providing such methods, the user supplied code is "decay agnostic" (for any smooth decay function): all the logic for when and how to merge and scale can be handled by the streaming system in response to a user specifying the desired decay function in an appropriate high-level language. Likewise, the sliding window approach is "decay agnostic", without any smoothness restrictions, and can even be supplied a decay function at runtime. It remains a challenge to develop these ideas and fully integrate them into a real system.

8 Related Work

There has been considerable study of computing decay functions over data streams in recent years. However, this work typically does not address out-of-order arrivals. The foundational work on providing guaranteed accurate answers to aggregates with sliding window decay came in the work on Exponential Histograms (EH) [14] and Deterministic Waves [16]. Both algorithms track counts and sums in sliding windows by keeping $O(\frac{1}{\epsilon}\log \epsilon N)$ counters. The EH approach can also work for aggregates that satisfy a certain set of conditions, and in particular that randomized "sketch summaries" (such as the AMS sketch [3] and Count-Min sketch [12]) can replace the counters. However, this approach blows up the size as a function of accuracy ϵ: keeping $O(\frac{1}{\epsilon}\log \epsilon N)$ sketches gives a total space bound of $\Omega(\frac{1}{\epsilon^3})$, which is impractical for small values of ϵ. Moreover, EH and Waves do not allow for out-of-order arrivals: the algorithms rely explicitly on packing together fixed numbers of items into each bucket of a histogram. Out-of-order arrivals overflow old buckets, so that the space and accuracy guarantees no longer hold.

Algorithms with approximation guarantees for out-of-order, or asynchronous, stream aggregates were designed by Busch *et al.* [8], who gave randomized algorithms for sliding window counts, and a randomized $O(\frac{1}{\epsilon^2}\log W)$ space algorithm for approximate sliding window

quantiles. Here, we show that randomness is unnecessary, and the problem can be solved deterministically with similar bounds. Recently, Busch and Tirthapura [7] gave a deterministic algorithm for the asynchronous sliding window count problem that uses $O(\frac{1}{\epsilon} \log W \log N)$ space; here we tighten this to $O(\frac{1}{\epsilon} \log W \log \epsilon N)$. [7] did not consider quantiles and heavy-hitter queries that we do here, and further, they focused exclusively on sliding window decay. Other approaches in the data stream literature for dealing with out-of-order arrivals are heuristic, involving buffering [1], load shedding [5], and punctuations [28].

There is much work on other specific problems in the sliding window decay model—these usually do not tolerate out-of-order arrivals. Babcock *et al.* [6] study tracking variance and k-median clustering, which cannot be solved using EH. Arasu and Manku [4] and Lee and Ting [19] have given improved bounds for quantiles and heavy hitters, respectively. These approaches rely on specific properties of the chosen aggregate, and exclude the possibility of late arrivals for similar reasons to the EH case: they rely on careful bucketing in the knowledge that no subsequent items which belong in the same bucket will be seen. Qiao *et al.* [26] describe heuristics for tracking time-varying distributions, in contrast to the guarantees we focus on.

Exponential decay has also attracted some interest due to its simplicity. For simple counts, exponential decay is virtually folklore, so for methods based on counts that are linear functions of the input, such as randomized sketches, the ability to apply exponential decay and out-of-order arrivals follows almost immediately. For other summaries, this is not so immediate: Manjhi *et al.* [20] carefully prove variations of Lossy Counting [21] and Frequent Elements [23] algorithms can track exponential decay in space $O(\frac{1}{\epsilon})$. Aggarwal [2] shows how to draw a sample with approximately exponential decay on sequence numbers. Again, these do not support out-of-order arrivals or the ability to merge summaries, in contrast to the methods we describe here.

The work of Cohen and Strauss [9] gave strong motivations for looking at decay functions other than sliding window and exponential decay. They introduced a variety of techniques for computing counts and sums under these decays. Taking inspiration from these, we extend to more general holistic aggregates, and out-of-order arrivals. We summarize the most related results in Table 1, and list the results from this paper for comparison. We observe that for many problems, our results are the first in the asynchronous model. The price paid for allowing out-of-order arrivals is small compared to the in-order case: typically, only a logarithmic factor of overhead.

9 Concluding Remarks

We have considered the problem of computing aggregates under decay functions over out-of-order streams. We have shown a variety of solutions for different classes of decay functions which are all fully deterministic, with precise space and time guarantees. Experimentally, we saw that exponential decay can be accommodated for quantiles and heavy hitters with almost no additional cost. Sliding window and polynomial decay come at a somewhat higher price, but still it is possible to process large streams accurately at high throughput: hundreds

of thousands of updates per second. It is open to improve these results, especially those for sliding windows. Our methods give a framework for a variety of aggregates. Some of the approaches, such as value-division, can be applied to arbitrary summaries satisfying certain properties (in this case, able to scale and merge the summary). It remains to fully understand which aggregates can be accurately approximated under this challenging model of decay.

References

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: a data stream management system. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, page 666, 2003.

[2] C. C. Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *Proceedings of the International Conference on Very Large Data Bases*, pages 607–618, 2006.

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *JCSS: Journal of Computer and System Sciences*, 58:137–147, 1999.

[4] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of ACM Principles of Database Systems*, pages 286–296, 2004.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of ACM Principles of Database Systems*, pages 1–16, 2002.

[6] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *Proceedings of ACM Principles of Database Systems*, 2003.

[7] C. Busch and S. Tirthapura. A deterministic algorithm for summarizing asynchronous streams over a sliding window. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, 2007.

[8] C. Busch, S. Tirthapura, and B. Xu. Sketching asycnhronous streams over sliding windows. In *ACM Conference on Principles of Distributed Computing (PODC)*, 2006.

[9] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *Proceedings of ACM Principles of Database Systems*, 2003.

[10] G. Cormode, F. Korn, S. Muthukrishnan, T. Johnson, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 35–46, 2004.

[11] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In *Proceedings of ACM Principles of Database Systems*, 2006.

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

[13] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *Proceedings of ACM Principles of Database Systems*, 2005.

[14] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[15] P. Flajolet and G. N. Martin. Probabilistic counting. In *IEEE Conference on Foundations of Computer Science*, pages 76–82, 1983. Journal version in *Journal of Computer and System Sciences*, 31:182–209, 1985.

[16] P. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. *Theory of Computing Systems*, 37:457–478, 2004.

[17] J. Hershberger, N. Shrivastava, S. Suri, and C. Toth. Adaptive spatial partitioning for multidimensional data streams. In *ISAAC*, 2004.

[18] Internet traffic archive. `http://ita.ee.lbl.gov/`.

[19] L.K. Lee and H.F. Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proceedings of ACM Principles of Database Systems*, pages 290–297, 2006.

[20] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *IEEE International Conference on Data Engineering*, pages 767–778, 2005.

[21] G.S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the International Conference on Very Large Data Bases*, pages 346–357, 2002.

[22] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of ICDT*, 2005.

[23] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.

[24] S. Muthukrishnan. Data streams: Algorithms and applications. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[25] Cisco NetFlow. More details at `http://www.cisco.com/warp/public/732/Tech/netflow/`.

[26] L. Qiao, D. Agrawal, and A. El Abbadi. Supporting sliding window queries for continuous data streams. In *Statistical and Scientific Database Management (SSDBM)*, 2003.

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

[28] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in countinuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, May 2003.