Time-Decaying Aggregates in Out-of-order Streams

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

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

Srikanta Tirthapura*
Iowa State University
snt@iastate.edu

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, data tuples pertaining to recent events are typically more important than older ones. This can be formalized via time-decay functions, which assign weights to data based on the age of data. Decay functions such as sliding windows and exponential decay have been studied under the assumption of well-ordered arrivals, i.e., data arrives in non-decreasing order of 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. Existing techniques such as Exponential Histograms and Waves are unable to handle out-of-order streams. We give the first deterministic algorithms for approximating these aggregates under popular decay functions such as sliding window and polynomial decay. We study the overhead of allowing out-of-order arrivals when compared to well-ordered arrivals, both analytically and experimentally. Our experiments confirm that these algorithms can be applied in practice, and compare the relative performance of different approaches for handling out-of-order arrivals.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures

General Terms: Algorithms

Keywords: Asynchronous Data Streams, Out-of-order Arrivals

1. INTRODUCTION

The challenge of observing, processing and analyzing massive streams of data is now 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 computation for analysis must preferably be done online as the stream is observed in whatever order it arrives (to alleviate worsening network traffic problems, exploit arbitrage opportunities in financial markets, etc).

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 a set of recent events and ignores older ones [15, 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 [22, 4]. 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 is due in part to the relative simplicity of algorithms to implement exponential decay [12]. 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 timestamp order. 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 re-

*Supported in part by NSF grant CNS-0520102, as well as by ICUBE, Iowa State University

cent elements on an out-of-order stream, the elements with the W *greatest timestamps* should be considered. In contrast, the usual definition of a sliding window [15, 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 [27], or load shedding when no other option is left [4]. However, such approaches may require too much overhead to put the stream in sorted order, apply to certain special cases, or lose accuracy guarantees due to dropped tuples, respectively. Therefore, we need techniques to summarize and analyze massive streaming data that are resilient to data arriving in arbitrary orders. Moreover, we need general approaches that allow aggregates under a variety of decay functions to be incorporated into streaming systems, to automate the handling of different decay functions from high level specifications.

The main contributions of this paper are as follows:

- We present the first space- and time-efficient algorithms for summarizing out-of-order streams to answer queries for key holistic aggregates such as *quantiles* and *heavy-hitters (frequent elements)* under different decay functions including *sliding window decay* and *polynomial decay*.
- Our algorithms provide deterministic guarantees on the quality of answers 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 $(\phi + \epsilon)$. Similar guarantees are provided for range queries and heavy hitters. These guarantees hold independent of the amount of "disorder" in the streams. The space taken as well as the time per item are both poly-logarithmic in the input size.
- We present an experimental evaluation of the algorithms discussed. We analyze the space used and the processing time per item. Our experiments show these algorithms can process hundreds of thousands of updates per second.

To our knowledge, this is the first work on estimating these aggregates over polynomial and general decay functions even on streams with well-ordered arrivals.

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

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: x_i could be the destination address, and t_i the time of observation of the packet; or, x_i could 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). We do not discuss the mapping of the raw data to $\langle x_i, t_i \rangle$ tuples further, but instead assume that such tuples can be easily extracted. All the methods in this paper 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.

The "current time" is denoted by the variable t. It is assumed that all times are non-negative integer values. Since there may be 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, and none for another timestamp.

We 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. This paper focuses on time-based semantics, since many of the techniques apply to tuple-based semantics.[1]

2.2 Decay Functions

A decay function takes the age of an item, and returns the weight for this item. A function $g()$ is considered 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 $g(a_1) \leq g(a_2)$

Some popular decay functions are:

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 can be computed for sums and counts. This special case is studied in more detail in [12].

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 $\alpha > 0$ ($(a+1)$ is used 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 abrupt or too gradual 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. In this work, we consider general decay functions with particular focus on sliding windows and polynomial decay.

[1]For example, for a sliding window of size W, the algorithms can be used to find a time t such that the number of items arriving between t' and t is (approximately) W, and so reduce tuple-based to time-based semantics.

2.3 Time-decayed aggregates

To sharpen the focus, we study a few popular aggregates (namely, 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; Section 7 discusses how the methods in this paper apply more broadly to other aggregates. We now define the time-decayed aggregates. These definitions are natural 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 $D(t) = \sum_i g(a_i)$ (referred to as D when t is implicit).

Our methods will, as a side effect, need to approximate $D(t)$.

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

$$\{p : \sum_{\{i:x_i=p\}} g(a_i) \geq \phi D\}.$$

For $g(a) = 1$ for all a (i.e., 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. It is the presence of time decay which provides the challenge in handling out-of-order arrivals.

Each item x_i is assumed to be an integer in the range $[1 \dots U]$. It is easy to verify that an exact solution 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 [15]. Computing the quantiles exactly even without decay requires space linear in the input size [23], and likewise for exact tracking of heavy hitters [21]. Therefore, approximation in the answers must be tolerated in order to make the resource requirements manageable. These are defined as follows:

Definition 4. For $0 < \epsilon < \phi \leq 1$, the *$\epsilon$-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 *$\epsilon$-approximate decayed ϕ-heavy hitters problem* is to find a set of items HH_ϕ which satisfies

$$HH_\phi \subseteq \{p : \sum_{\{i:x_i=p\}} g(a_i) \geq (\phi - \epsilon)D\}$$
$$\text{and } HH_\phi \supseteq \{q : \sum_{\{i:x_i=q\}} g(a_i) \geq (\phi + \epsilon)D\}.$$

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

Example. Consider the input stream $\langle y, 2\rangle, \langle x, 3\rangle, \langle y, 1\rangle$ with three items; x with a timestamp 3, and two copies of item y, with timestamps 2 and 1. Assume that all these items have arrived at time 3, before any query was posed. Consider also a polynomial decay function $g(a) = (1 + a)^{-1}$. If a decayed ϕ-heavy hitters query with $\phi = \frac{1}{2}$ is posed at time 3, only x should be returned, since x has decayed weight 1 while y has decayed weight $\frac{1}{2} + \frac{1}{3} = \frac{5}{6}$, so that only x has (decayed) relative weight of $\frac{1}{2}$ or greater. 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 $\frac{1}{3} + \frac{1}{4} = \frac{7}{12}$, so that only y has (decayed) relative weight of $\frac{1}{2}$ or greater. □

Thus, even *a priori* knowledge of the decay function and query to be posed does not allow a single result to be precomputed and stored; instead sufficient information must be retained to answer the query whenever it is posed.

2.4 Semantics of Late Arrivals

A late arriving tuple may require a previously reported answer to change. However, revising these obsolete answers is problematic because reconstructing the value of an answer reported in the past and revising it to the correct answer based on the late arrival would effectively mean "rewinding" to a previous state of the data structure; this requires linear space, as formalized in the lemma below. One could try to estimate such values using the data structure at the current time, but for very late arrivals the approximation guarantees could be significantly bad (out-of-order arrivals which are only slightly delayed can be corrected for with better accuracy). Therefore, when a late tuple arrives, we choose to not correct for any previous aggregate computations rendered incorrect by it. 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 no late tuples are to be expected; this most honestly reflects the observer's knowledge.

LEMMA 1. *Correcting previously reported answers requires $\Omega(N)$ space, where N the number of items in the stream.*

PROOF SKETCH. Consider answering a sliding window sum over a small sliding window of size, say, $W = 1$ time step. Correcting a previously reported answer with less than $\epsilon = \frac{1}{2}$ relative error given an out-of-order arrival with timestamp t means recovering whether there was an arrival during time t or not. This can be used to encode a bitstring of length N, and therefore $\Omega(N)$ bits are required (even allowing randomization). Similar reductions show the hardness for other aggregate queries and decay models. □

3. SLIDING WINDOW

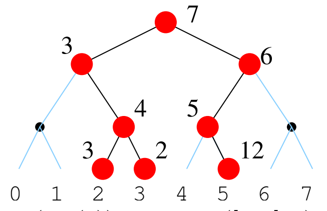
In this section we show a deterministic summary which allows us to answer queries for both quantiles and heavy hitters under sliding windows, where updates may arrive out-of-order.

3.1 Approximate Window Count

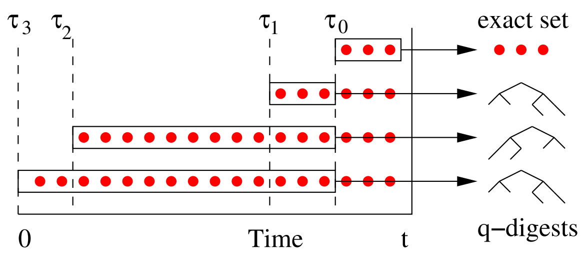
We first introduce a data structure which solves the simpler problem of tracking the decayed count of items within a sliding window as they arrive in an arbitrary order. This question has been studied in prior work explicitly [26, 7] and implicitly [11]. Our solution here meets the best existing space bounds for this problem [11, 7], and does so using a relatively simple construction which can subsequently be extended to answer other aggregates.

Given window size w (specified at query time) and a stream $\langle x_i, t_i\rangle$, the aim is to approximate $D_w(t) = |\{i : t - t_i < w\}|$ with ϵ relative error. The analysis assumes that each t_i is an integer in the range $[0 \dots W - 1]$, and analyzes the space complexity as a function of W. Equivalently, W is an upper bound on the window size w. For simplicity, 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. The solution makes use of the q-digest data structure due to Shrivastava *et al.* [25] which has the following main properties:

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 $[0 \dots W - 1]$. 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, that is, $\mathrm{r}(q) = \sum_{i<q} f_i$. The data structure maintains an appropriately defined set of *dyadic ranges* $\subseteq [0 \dots W - 1]$ and their associated counts. A *dyadic range* is a range of the form $[i2^j \dots (i + 1)2^j - 1]$ 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



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

$[a \ldots b]$ can be uniquely partitioned into at most $2\log(b-a)$ dyadic ranges, with at most 2 dyadic ranges of each length. Based on a parameter $\gamma = \frac{\epsilon N}{\log_2 W}$, our realization of the q-digest has the following properties:

- Each range r has an associated count $c(r) \leq \gamma$ 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
$$c(\mathrm{par}(r)) + c(\mathrm{left}(\mathrm{par}(r))) + c(\mathrm{right}(\mathrm{par}(r))) \geq \gamma.$$
- 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 $q \in [0 \ldots W-1]$, the estimate the rank of q, denoted by $\hat{r}(q)$, is the sum of the counts of all ranges to the left of q, i.e. $\hat{r}(q) = \sum_{\{r=[l,h]:h<q\}} c(r)$. The estimate of the rank satisfies: $\hat{r}(q) \leq \mathrm{r}(q) \leq \hat{r}(q) + \epsilon N$. Similarly, given a query point q the estimate of f_q, the frequency of item q is $\hat{f}_q = \hat{r}(q+1) - \hat{r}(q)$, so that $\hat{f}_q - \epsilon N \leq f_q \leq \hat{f}_q + \epsilon N$.

It is shown in [25, 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. In pseudocode, we will refer to this as the QDINSERT routine. Periodically, the data structure can be pruned to ensure it meets its space bounds, by calling a QDCOMPRESS routine. An example q-digest is shown in Figure 1(a). □

Sliding Window Count Algorithm. Our solution to the sliding window count problem uses 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 sliding window count problem asks 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 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. Define $\alpha = \frac{9}{\epsilon}\log W$. The algorithm maintains many data structures $Q_0, Q_1, \ldots, Q_J$ (J defined below); see 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 the (roughly) $2^j \times \alpha$ most recent elements of the stream. As j increases, more elements are summarized by Q_j, so the absolute error of estimates provided by Q_j can increase since, if $\mathrm{rr}(t-w)$ is large, then to estimate $\mathrm{rr}(t-w)$ it suffices to use Q_j where j is large to stay within the relative error bound.

This use of the q-digest differs from the way it is employed in previous work [25] in the following way: while in [25], the upper bound on the count of a node in the q-digest, γ, increases with the number of elements being summarized, here in Q_j, $\gamma(j)$ is set to 2^j. 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 can increase beyond α so some ranges must be discarded. Informally, the α "most recent" ranges in Q_j are retained, and the rest discarded. More formally, periodically, a SWCOMPRESS routine is called which sorts the ranges within Q_j according to increasing order of right endpoints; ties are broken by putting smaller ranges first (this corresponds to a post-order traversal of the tree implicitly represented by the q-digest). The α rightmost elements in this sorted order are retained in Q_j.

Each q-digest $Q_j, j > 0$ tracks the minimum time τ_j, 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 $\tau \geq \tau_j$, an estimate $\hat{\mathrm{rr}}(\tau)$ of $\mathrm{rr}(\tau)$ is found from Q_j as the sum of the counts in all ranges $[l, h]$ in Q_j such that $l > \tau$. That is, $\hat{\mathrm{rr}}(\tau) := \sum_{\{r:(r=[l,h]\in Q_j)\wedge(l>\tau)\}} c(r)$.

Accuracy. $\hat{r}(\tau)$ overestimates $r(\tau)$, since it counts all updates more recent than τ, and some that are older. But the error in the estimate $\hat{r}(\tau)$ 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 $\gamma(j)\log W$. Thus if $\tau \geq \tau_j$, then:

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

It also follows that if Q_j is "full", i.e. the number of ranges in Q_j is the maximum possible, then $\mathrm{rr}(\tau_j) \geq \frac{\alpha}{3}\gamma(j) - \gamma(j)\log W$, from the second QD property and the $\log W$ ranges that can contain τ_j. Using $\epsilon < 1$ and the settings of α and $\gamma(j)$, it follows that:

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

Given window size w, the 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 $\hat{\mathrm{rr}}_\ell(t-w)$ using Q_ℓ. The accuracy guarantee can be shown as follows. If $\ell = 0$, then the algorithm has returned the exact answer. Otherwise, from (1):

Algorithm 3.1: SWUPDATE(t)

```
Input: time stamp $t$
$Q_0 \leftarrow Q_0 \cup \{t\}$
if $|Q_0| > \alpha$
  then $T \leftarrow \min t \in Q_0$
       delete $T$ from $Q_0$
       for $j = 1$ to $J$
         do if $T > \tau_j$
           then QDINSERT($Q_j, T$)
```

Algorithm 3.2: SWCOMPRESS()

```
for $j \leftarrow 1$ to $J$
  do recompute $\tau_j$
     for each $(r, c(r)) \in Q_j$
       do if $\max(r) \le \tau_j$
         then delete $(r, c(r))$ from $Q_j$
     QDCOMPRESS($Q_j$)
```

Algorithm 3.3: SWQUERY(w)

```
Input: window size $w$
if $t - w \ge \tau_0$
  then return $(|\{\tau \in Q_0 | \tau > t - w\}|)$
  else $\ell = \arg\min_i(\tau_i \le t - w)$
       return $(\hat{\mathrm{rr}}_\ell(t - w))$
```

Figure 2: Pseudocode for Sliding Window Count

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

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

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

Thus the relative error

$$\frac{|\hat{\mathrm{rr}}_\ell(t-w) - \mathrm{rr}(t-w)|}{\mathrm{rr}(t-w)} \le \frac{2^\ell \log W}{2^\ell \epsilon^{-1} \log W} \le \epsilon$$

Pseudocode in Figure 2 shows the routines for inserting a new time stamp(SWUPDATE), compressing to ensure the space bounds (SWCOMPRESS), and answering a count query (SQQUERY).

Space and Time Complexity. 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 needed is Q_J for J given by $\frac{2^J}{\epsilon}\log W \ge N$, 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 $O(\log J) = O(\log\log(\epsilon N))$ via 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, it suffices to use the same general structure as the algorithm for the count, but instead of just keeping counts within each q-digest, more details are kept 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 $r(w,x) = |\{i | x_i \le x, t - t_i \le w\}|$. The approximate sliding window range query problem, given (w,x) with $0 \le w < W, 0 \le x < U$, is to return an estimate $\hat{r}(w,x)$ such that $|\hat{r}(w,x) - r(w,x)| \le \epsilon D_w(t)$, where $D_w(t) = |\{i : t - t_i < w\}|$.

This problem statement is crucial for the subsequent reductions. It is convenient to think of these as range queries over two-dimensional (item, time) pairs which arrive in a stream. Observe that the required approximation quality is $\epsilon D_w(t)$, rather than $\epsilon r(w,x)$. Estimating $r(w,x)$ with approximation error $\epsilon r(w,x)$ is not possible in small space, since the problem of maintaining the maximum of elements within a sliding window, which is known to require space linear in the size of the stream [15], can be reduced to the estimation of $r(w,x)$ with a relative error of ϵ. 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 (i.e. additive error of ϵN [17]) so a modified approach is needed.

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 J q-digests $Q_0, Q_1, \ldots, Q_J$, much as in the sliding window count case above. Each of these orders data along the "time" dimension—we call these the "time-wise" q-digests. Within $Q_j, j > 0$ the threshold $\gamma(j) = 2^{j-1}$. Within each range $r \in Q_j$ another q-digest which summarizes data along the value-dimension is kept. These are called the "value-wise" q-digests. This approach of considering each dimension in turn is fairly standard, and similar algorithms with different accuracy guarantees are given in [17]. So we only briefly describe two alternate realizations of this idea which provide the guarantees required in Definition 5 in slightly different bounds.

Eager merge algorithm. The value-wise q-digests within Q_j are maintained with $\gamma(j) = \frac{2^{j-1}}{\log U}$. 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.

Estimation: To estimate $r(w,x)$, our algorithm uses Q_ℓ where ℓ is the smallest integer such that $\tau_\ell \le t - w$. Within Q_ℓ, it finds at most $\log W$ value-wise q-digests to query based on a dyadic decomposition of the range $(t-w, t]$, and each of these is queried for the rank of x. Finally, the estimate $\hat{r}(w,x)$ is the sum of these results.

Accuracy: The error of the estimate has two components. Firstly, within the time-wise q-digest Q_ℓ error of up to $2^{\ell-1}\log W$ is incurred since the number of elements within the timestamp range may be undercounted by up to $2^{\ell-1}\log W$. Next, within each value-wise q-digest, error of up to $\frac{2^{\ell-1}}{\log U}\log U = 2^{\ell-1}$ is incurred. Since as many as $\log W$ value-wise q-digests may be

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 $2\cdot 2^{\ell-1}\log W = 2^{\ell}\log W$. Choosing $\alpha = \frac{3}{\epsilon}\log W$ ranges within each each Q_j, and using a similar argument to the count algorithm, gives $D_w \geq \mathrm{rr}(\tau_{\ell-1}) > \frac{2^{\ell}\log W}{\epsilon}$. Thus the error in the 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 $O(\log W\,\mathrm{rr}(\tau_j)) = O(\frac{1}{\epsilon}2^j\log^2 W)$, 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 this fraction of the total count requires $O(1)$ space. 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 time-wise 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 $O(\log(\frac{\epsilon N}{\log W})\log W(\log\log U))$ $= O(\log(\epsilon N)\log W(\log\log U))$. $\Box$

Defer Merge algorithm. The defer merge version uses a similar idea, of time-wise q-digests Q_j, each node of which contains a value-wise q-digest. It uses the same number and arrangement of time-wise q-digests, but a different arrangment of value-wise structures. Instead of inserting each update in all value-wise q-digests that summarize time ranges in which it falls, it is inserted in only one, corresponding to the node in the time-wise structure whose count is incremented 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 $O(\frac{\log U}{\epsilon})$.
Estimation: To answer queries $r(w,q)$, τ_ℓ is found based on w and query Q_ℓ as before. All value-wise summaries within Q_ℓ which correspond to items arriving within the time window $(t-w,t]$ are merged together (this merging is deferred to query time, as opposed to the above "eager merge" approach, which computes the result of this merging at insertion time). The query x is then posed to the resulting q-digest.
Accuracy: Again, error $2^{\ell-1}\log W$ is incurred from uncertainty in Q_ℓ. 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: The space required is bounded 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 trading off a factor of $\log W$ for one of $\frac{1}{\epsilon}$ compared to the above eager-merge version). Each insertion, requires an insertion into the time-wise q-digest for each Q_j, 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 $O(\log(\frac{\epsilon N}{\log W})(\log\log W + \log\log U))$. $\Box$

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.*

3.3 Reduction 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, the data structure for range queries is created with accuracy parameter $\frac{\epsilon}{2}$. To answer a query for the approximate ϕ-quantile, an approximation $\hat{D}_w$ of D_w is first computed using the time-wise q-digests, through the results of Theorem 1. Then the smallest x such that $\hat{r}(w,x) \geq \phi\hat{D}_w$ is found by binary search. Observe that such an x satisfies the requirements for being an approximate ϕ-quantile: $|\hat{D}_w - D_w| \leq \frac{\epsilon}{2}D_w$, 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 by posing 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. *Approximate sliding window quantile and heavy hitters with out-of-order arrivals can be answered in the same update and space bounds as Theorem 2.*

This lets window size $w \leq W$ to be specified at query time; if instead it is fixed to W tuples and only the q-digest for the appropriate τ_j is kept, a factor of $O(\log(\frac{\epsilon N}{\log W}))$ is saved in the bounds.

4. POLYNOMIAL AND GENERAL DECAY

We describe two methods to handle decay functions other than sliding window decay, in particular, polynomial decay. The first method applies to *any* decay function by reducing a query for decayed sums/counts to multiple window queries (à la [9]); we describe how to execute this reduction efficiently. The second method, "Value-Division", is potentially more efficient, and, although less general, applies to a broad class of "smooth" decay functions (defined formally below) which includes polynomial decay. We focus the discussion on polynomial decay for ease of exposition.

4.1 Reduction to Multiple Sliding Windows

We first outline a generic approach for decayed aggregates by reduction to multiple queries over our proposed sliding window summary. This is based on the observations in [9] for computing decayed sums and counts. Consider an arbitrary decay function $g(a)$. The (decayed) rank of any item x can be written as the sum

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

This shows that in the space bounds given by Theorem 2 arbitrary decay functions can be applied, due to inherent discretization of the decay function at each timestep. Because the rank of x in window j is approximated with error ϵD_j, summing all counts preserves the relative error:

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

Note that this algorithm as stated is not time efficient due to the current need for multiple queries. Yet it is straightforward to extend it to run more quickly, by more cleverly maintaining the contents of the data structure. It is not necessary to query all possible time values, since the data structure only stores explicit information about a limited number of time stamps, and queries about sliding windows with other timestamps will give the same answers as queries

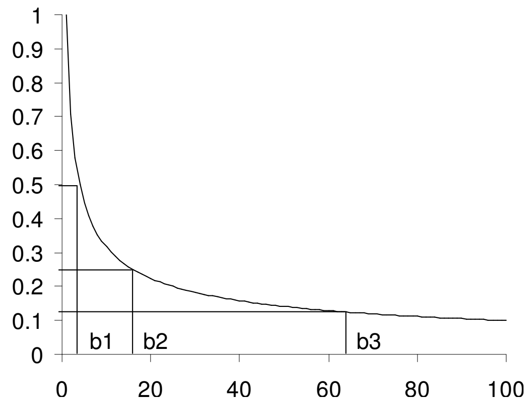


Figure 3: Choice of boundaries in value-division approach

on some timestamp stored in the data structure. Thus the sum need only be evaluated at timestamps stored in the data structure, rather than all possible timestamps.

Using the reductions in Section 3.3, quantiles and heavy hitters can also be found using the same data structure. Note that these are very strong results: not only can item arrivals be handled in completely arbitrary orders, but also any arbitrary decay function can be applied 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 3. *Decayed range sum, heavy hitter and quantile queries can be answered on out-of-order arrivals in the bounds stated in Theorem 2. Queries take time linear in the data structure size, i.e.* $O\left(\frac{1}{\epsilon}\log U \log W \log \epsilon N \min(\log W, \frac{1}{\epsilon})\right)$.

4.2 Value-Division

The above algorithm is quite general, but the space bounds and time cost may be improved by using an alternate approach. We generalize the technique of Cohen and Strauss for decayed sums and counts [9] to apply to quantiles and heavy hitters. In fact, these results apply to a broad range of aggregates, comprising any aggregate which has a summary such that two summaries can be merged to get a summary of the union of the inputs, and scaling a summary obtains 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 aggregates 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 decay is not smooth since it is not continuous. Polynomial decay is smooth, as is exponential decay (the definition holds with equality).

Value divisions. Given a smooth decay function g, define the set of boundaries on ages, b_i, so that $g(b_i) = (1+\theta)^{-i}$ for θ chosen later. The value division keeps a small number of summaries of the input. Each summary s_j corresponds 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}. Figure 3 illustrates the decay function $g(a) = (1+a)^{-1/2}$, so for $\theta = 1$, boundaries are at $b_1 = 3$ ($g(b_1) = \frac{1}{2}$), $b_2 = 15$ ($g(b_2) = \frac{1}{4}$), and $b_3 = 63$ ($g(b_3) = \frac{1}{8}$).

The boundaries define the summary time intervals: for all boundaries b_i at time t, at most one summary s_j is permitted such that

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

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

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

(i.e. both summaries fall between adjacent boundaries), summaries s_j and s_{j+1} are merged to 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, the item is included in the unique summary which covers t_i. This works because the choice of boundaries is independent of the arrivals.

THEOREM 4. *Given a linear summary algorithm, a* $(1+\theta)$ *accurate answers to (polynomial) decay queries can be found by storing* $O(\log_{1+\theta} g(t))$ *summaries. Updates take amortized time* $O(\log g(t))$.

PROOF. We first show that an accurate summary can be built 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}:

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

and so

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

(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, treating each item in s_j as if it arrived at time t_j only affects 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. $(t - t_{j+1}) \geq b_i \geq (t - t_j)$). 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, it follows that, at the time of formation $g(t - t_{j+1})/g(t - t_j) \leq (1+\theta)$. If g is smoothly decaying, this remains true for all times $T > t$. Let $a = (t - t_{j+1})$ and $A = (t_j - t_{j+1})$, 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, which shows that treating all items summarized by one summary 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, 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. The query is answered 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 based on a cut off point W beyond which $g(a > W) = 0$). At any instant, each summary either falls between two boundaries, or crosses a boundary. There is at most one summary 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 $O(\frac{1}{\theta}\ln(\frac{1}{g(t)}))$. $\square$

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 [12], and so can be used with polynomial decay. The total space required is therefore $O(\frac{1}{\theta}\ln(1/g(t)) \cdot \frac{\log U}{\beta}) = O(\frac{\alpha}{\theta\beta}\log U \log t)$ for polynomial decay. The total approximation error is, in the worst case, $(\theta + \beta)D$. In order to guarantee overall error of ϵD, set $\theta + \beta = \epsilon$; the space is minimized by $\theta = \beta = \epsilon/2$, giving $O(\frac{\log U}{\epsilon^2}\log t)$ overall. The time cost is $O(\log t)$ amortized per update.

The space used is comparable to the general bound 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 boundaries are created based on $g(a) = (1+a)^{-2}$, they are located at $a = 1, (1+\theta)^{1/2}, (1+\theta), (1+\theta)^{3/2} \ldots$. This is a superset of the boundaries that would have been 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:

LEMMA 2. *Given the results of running the value-division algorithm with boundaries b_i based on decay function g and parameter θ, at query time, a $(1+\theta')$ accurate answer can be found for any smooth decay function g', provided $\forall i. g(b_i)/g(b_{i+1}) \leq (1+\theta')$.*

Thus, boundaries can be set based on a particular function g and θ value, a new function g' can be specified that is "weaker" than g (decays less quickly), getting a guarantee with a θ' that is better than the original θ. Equally, a g' can be specified that is stronger than g (decays faster), and a result obtained 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 $\theta' = 2\theta + \theta^2$.

5. 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 their space usage and performance in practice. We compare polynomial and sliding window decay functions to the cost of aggregate computations with no decay function (which can ignore timestamp and hence arrival order).

5.1 Experimental Setup

We implemented our core methods based on q-digests 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 are roughly the same size: 12 bytes per node in our implementation (input $\langle x_i, t_i \rangle$ pairs are 8-12 bytes). For the sliding window algorithms, we compared the eager merge and defer merge strategies. On our data sets the defer-merge approach was often 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, 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 (`http://ita.ee.lbl.gov/`), 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.

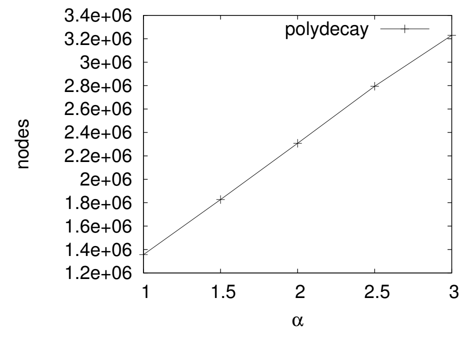
5.2 Space Usage

To understand 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 total error ϵ fixed at 0.1 in Figure 4(a). 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 4(b) plots space against ϵ for these two approaches. It shows that the value division approach can be significantly more space-efficient than sliding windows, especially with small values of ϵ (results using the flow data are shown).

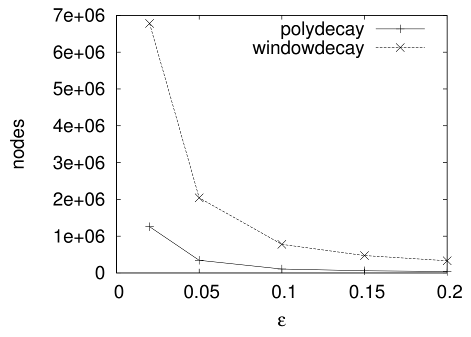
Recall that for both data sets, the total input size is 5 million items. Thus, for moderate values of ϵ, the space used by our approximate algorithms (especially value-division) is significantly less than that of the input, sometimes by an order of magnitude or more. As ϵ decreases, the space rises sharply, 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 timewise q-digests Q_j. Here, the value-division approach (not shown) is guaranteed to use fewer nodes that the number of input tuples, and so is a competitive alternative.

5.3 Performance

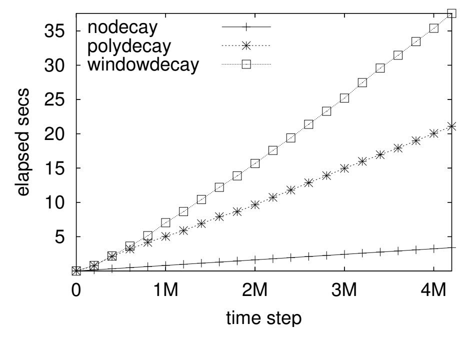
Figure 4(c) compares the time (in seconds) taken to update the data structure for the polynomial and sliding window decay functions as well as the q-digest (no-decay) as a baseline, at increasing timestamps using World Cup data. With flow data (not shown), the curve ordering was the same but the smaller amount of disorder enabled the value division approach to achieve performance closer to that of no decay. Figure 4(d) 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



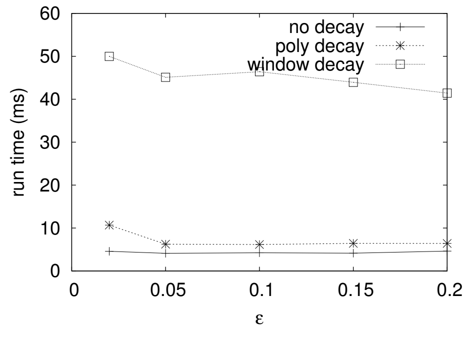
(a) poly decay: space vs α ($\epsilon = 0.1$, flow data)



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



(c) performance vs timestamp (World Cup)



(d) performance vs ϵ (flow)

Figure 4: Experimental results: (a) space for polynomial decay w.r.t. α (b) space for polynomial and window decay (c) time for processing World Cup data (d) time for processing 5M flow records

the summaries in order from the most recent. (For small ϵ, there are more summaries to search through.) An implementation with a dynamic index on the timestamps would improve the time cost. Similarly, on highly out-of-order data, the running time increased due to inserting into more levels in the sliding window algorithms.

Overall, our methods seem suited to high-speed data stream systems, since they are capable of processing up to a hundred thousand items per second on an off-the-shelf system. The extra flexibility and sophistication that sliding windows provide is obtainable, at a premium (constant factors in time and space in our experiments).

6. RELATED WORK

There has been considerable recent research on computing decayed aggregates over data streams. However, this work typically does not address out-of-order arrivals. The first algorithms for providing guaranteed accurate answers to aggregates using sliding window decay are the Exponential Histograms (EH) [15] 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 [2] and Count-Min sketch [13]) can replace the counters. However, this approach blows up the space as a function of accuracy ϵ: keeping $O(\frac{1}{\epsilon}\log \epsilon N)$ sketches of size $\Omega(\frac{1}{\epsilon^2})$ gives a total space bound of $\Omega(\frac{1}{\epsilon^3})$, which is impractical for most values of ϵ. Recently, Smooth Histograms (SH) [6] have been proposed to improve exponential histograms by taking advantage of properties of the aggregate function. But importantly, EH, SH 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.

There is much work on complex aggregates in the sliding window decay model—these usually do not tolerate out-of-order arrivals. Qiao *et al.* [24] describe heuristics for tracking histograms, in contrast to the guarantees we provide. Babcock *et al.* [5] study tracking variance and k-median clustering, which cannot be solved using EH. Arasu and Manku [3] and Lee and Ting [19] have given improved bounds for quantiles and heavy hitters, respectively. They rely on specific properties of the chosen aggregate, and exclude the possibility of late arrivals for similar reasons to the EH case: they perform careful time bucketing assuming that no subsequent items which belong in the same bucket will be seen.

Algorithms with approximation guarantees for out-of-order, or asynchronous, stream aggregates were designed by Tirthapura *et al.* [26], who gave randomized algorithms for estimating the sliding window count, 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 problems 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, as we do in this work, and further, they focused solely 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 [4], and punctuations [27].

Exponential decay has attracted 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. We recently showed how to maintaining aggregates using data structures tailored to exponential decay in [12]; these techniques easily allow out-of-order arrivals.

The work of Cohen and Strauss [9] gave strong motivations for looking at decay functions other than sliding window and exponential decay. They introduced Cascaded Exponential Histograms (CEH) and Weight-Based Merging Histograms (WBMH) for computing counts and sums under these decays (subsequently, some improvements from $O(\log n \log\log n)$ to $O(\log n)$ for WBMH have been proposed by Kopelowitz and Porat [18]). Taking inspiration from these, we extend to more general holistic aggregates, and out-of-order arrivals. 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 asymptotically small compared to the in-order case: typically, a logarithmic factor of overhead.

7. EXTENSIONS

We briefly note some extensions to our techniques:

• **System issues.** Our methods are quite general and apply to any aggregate which has a summary such that two summaries can be merged to generate a summary of the union of the inputs; or scaled to obtain a summary of the linearly scaled input. There are referred to as "linear summaries". This incorporates most "sketch" algorithms since they are typically based on maintaining arrays of counts or sums [2, 13]. Streaming systems support user-defined decayed aggregate functions (UDAFs) by requiring the user to specify a small number of subroutines [10, 8]. Additionally supporting decay (e.g., via a `DECAY BY` clause) requires only a few extra routines such as `Scale` and `Merge`, for the DSMS to multiply the summary by a scalar when needed and merge two summaries when needed, respectively; the logic for *when* to scale and merge can be handled by the streaming system. Depending on which underlying approach is used (sliding windows versus value-division), the supplied decay function can be quite general and, in some cases, supplied at query time. We have shown both analytically and experimentally that performance depends on which approach is used; thus, the system can optimize, given user requirements.

• **Distributed observations.** In the distributed setting, separate observers see independent streams and must summarize the union of the input. In the sliding window case, summaries can be merged and compressed, to obtain the same space bounds as the centralized case (unlike EH/Waves [15, 16]). In the value-division approach summaries can be merged accurately, but the space is not bounded.

• **Duplications and Deletions.** Data quality issues mean that streams may contain duplicate arrivals which should be counted once, or retractions of prior updates. For duplicates, replacing counters with "approximate count distinct sketches" will suppress duplicates [14]. This blows up the space; alternate approaches are based on the technique of "distinct sampling" [16, 26]. Deletions can be handled by replacing q-digests with randomized sketches which can tolerate deletions, such as the Count-Min sketch [13]

8. SUMMARY

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 it is possible to process large streams accurately at high throughput (hundreds of thousands of updates per second) for sliding window and polynomial decay. 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, ability to scale and merge the summary). It remains to fully understand which aggregates can be accurately approximated under this challenging model of decay.

9. REFERENCES

[1] D. Abadi *et al.* Aurora: a data stream management system. In *SIGMOD*, 2003.

[2] 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.

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

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[5] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *PODS*, 2003.

[6] V. Braverman and R. Ostrovsky Smooth Histograms for Sliding Windows. In *FOCS*, 2007.

[7] C. Busch and S. Tirthapura. A deterministic algorithm for summarizing asynchronous streams over a sliding window. In *STACS*, 2007.

[8] S. Cohen. User-defined aggregate functions: bridging theory and practice. In *SIGMOD*, 2006.

[9] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *PODS*, 2003.

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

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

[12] G. Cormode, F. Korn, and S. Tirthapura. Exponentially Decayed Aggregates on Data Streams. In *ICDE*, 2008.

[13] 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.

[14] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *PODS*, 2005.

[15] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *SODA*, 2002.

[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] T. Kopelowitz and E. Porat. Improved Algorithms for Polynomial Time-Decay and Time-Decay with Additive error. In *ICTCS*, 2005.

[19] L.K. Lee and H.F. Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *PODS*, 2006.

[20] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *ICDE*, 2005.

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

[22] S. Muthukrishnan. Data streams: Algorithms and applications. In *SODA*, 2003.

[23] J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980.

[24] L. Qiao, D. Agrawal, and A. El Abbadi. Supporting sliding window queries for continuous data streams. In *SSDBM*, 2003.

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

[26] S. Tirthapura, C. Busch, and B. Xu. Sketching asycnhronous streams over sliding windows. In *PODC*, 2006.

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