Time-Decaying Sketches for Sensor Data Aggregation*

Graham Cormode
AT&T Labs–Research
180 Park Avenue
Florham Park, NJ 07932
graham@research.att.com

Srikanta Tirthapura
Dept. of ECE
Iowa State University
Ames, IA 50011
snt@iastate.edu

Bojian Xu
Dept. of ECE
Iowa State University
Ames, IA 50011
bojianxu@iastate.edu

ABSTRACT

We present a new sketch for summarizing network data. The sketch has the following properties which make it useful in communication-efficient aggregation in distributed streaming scenarios, such as sensor networks: the sketch is duplicate-insensitive, i.e. re-insertions of the same data will not affect the sketch, and hence the estimates of aggregates. Unlike previous duplicate-insensitive sketches for sensor data aggregation [26, 12], it is also time-decaying, so that the weight of a data item in the sketch can decrease with time according to a user-specified decay function. The sketch can give provably approximate guarantees for various aggregates of data, including the sum, median, quantiles, and frequent elements. The size of the sketch and the time taken to update it are both polylogarithmic in the size of the relevant data. Further, multiple sketches computed over distributed data can be combined without losing the accuracy guarantees. To our knowledge, this is the first sketch that combines all the above properties.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network Monitoring; H.2.8 [**Database Applications**]: Data Mining; H.3.4 [**Systems and Software**]: Distributed Systems

General Terms

Algorithms, performance, design, reliability, theory

Keywords

Sensor network, data aggregation, data stream processing, time decay, sliding windows, asynchronous streams, distributed streams, duplicate insensitive sketch

*The work of Tirthapura and Xu was supported in part by NSF grant CNS-0520102 and by ICUBE, Iowa State University

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODC'07, August 12–15, 2007, Portland, Oregon, USA.
Copyright 2007 ACM 978-1-59593-616-5/07/0008 ...$5.00.

1. INTRODUCTION

The growing interest in sensor networks has led to a variety of novel problems in distributed computing. Although sensors are increasing in computing ability and number, they remain constrained by the cost of communication: with limited batteries, communication is the primary drain on power and so the working life of a sensor network can be extended by algorithms which limit communication [23]. In particular, this means that although sensors may observe large quantities of information over time, they should return only small summaries of their observations. Ideally, we should be able to use a single compact summary that is flexible enough to provide estimates for a variety of aggregates, rather than using different summaries for estimating different aggregates.

The sensor network setting leads to several other desiderata. Because of the radio network topology, it is common to take advantage of the 'local broadcast' behavior, where a single transmission can be received by all the neighboring nodes. Here, in communicating back to the base station, each sensor opportunistically listens for information from other sensors, merges received information together with its own data to make a single summary, and announces the result. This multi-path routing has many desirable properties: appropriate merging ensures each sensor sends the same amount, a single summary, and the impacts of loss are much reduced, since information is duplicated many times (without any additional communication cost) [26, 12]. However, this duplication of data requires that the quality of our summaries remains guaranteed, no matter whether a particular observation is contained within a single summary, or is captured by many different summaries. In the best case the summary is *duplicate-insensitive* and *asynchronous*, meaning that the resulting summary is identical, irrespective of how many times, or in what order, the data is seen and the summaries are merged.

Lastly, we observe that in any evolving setting, recent data is more reliable than older data. We should therefore weight newer observations more heavily than older ones. This can be formalized in a variety of ways: we may only consider observations that fall within a *sliding window* of recent time (say, the last hour), and ignore (assign zero weight to) any that are older; or, more generally, use some arbitrary function $f()$ that assigns a weight to each observation as a function of its age [15, 11]. A data summary should allow such decay functions to be applied, and give us guarantees relative to the exact answer.

Putting all these considerations together leads to quite

an extensive requirements list. We seek a *compact*, *general purpose* summary, which can apply arbitrary *time decay functions*, while remaining *duplicate insensitive* and handle *asynchronous arrivals*. Further, it should be easy to *update* with new observations, *merge together* multiple summaries, and *query* the summary to give guaranteed quality answers to a variety of analysis. Prior work has considered various summaries which satisfy certain subsets of these requirements, but no single summary has been able to satisfy all of them. Here, we show that it is possible to fulfill all the above requirements sketch based on a a hash-based sampling procedure that allows a sampling procedure that allows a variety of aggregates to be computed efficiently under arbitrary decay functions in a duplicate insensitive fashion over asynchronous arrivals. In the next section, we describe more precisely the setting and requirements for our data structures.

1.1 Problem Formulation

Consider a data stream of observations seen by a single sensor $R = \langle e_1, e_2, \ldots, e_n \rangle$. Each observation e_i, $1 \le i \le n$ is a tuple (v_i, w_i, t_i, id_i), where the entries are defined as follows:

- v_i is a positive integer value, perhaps a temperature observation by the sensor.
- w_i is a weight associated with the observation, perhaps a number reflecting the confidence in it.
- t_i is the integer timestamp, tagged at the time e_i was created.
- id_i is a unique observation id for e_i.

This abstraction captures a wide variety of cases that can be encoded in this form. It is deliberately general; users can choose to assign values to these fields to suit their needs. For example, if the desired aggregate is the median temperature reading across all (distinct) observations, this can be achieved by setting all weights to $w_i = 1$ and the values v_i to be actual temperatures observed. The unique observation id id_i can be formed as the concatenation of the unique sensor id and time of observation (assuming there is only one reading per instant). We shall give other examples in the next section.

It is possible that the same observation appears multiple times in the stream, with the same id, value and timestamp preserved across multiple appearances — such repeated occurrences must not be considered while evaluating aggregates over the stream. Note that our model allows different elements of the stream to have different ids, but the same values and/or timestamps — in such a case, they will be considered separately in computing the aggregates.

We consider *asynchronous* streams, where the elements do not necessarily arrive in order of timestamps. Handling asynchrony is especially important because of multi-path routing, as well as the need to handle the union of sketches. Note that e_{i+1} is received after e_i, and e_n is the most recently received item. In the asynchronous case, it is possible that $i > j$, so that e_i is received later than e_j, but $t_i < t_j$. Most prior research on summarizing data streams containing timestamps (with the exception of [28, 8]) has focused on the case of *synchronous* streams, where the elements of the stream are assumed to arrive in the order of timestamps.

Definition 1.1. *A function $f(x)$ is a decay function if: (1) for every x, $f(x) \ge 0$ and (2) $f(x)$ is non-increasing with x, i.e. $x_1 \ge x_2 \implies f(x_1) \le f(x_2)$.*

The result of the decay function is applied on the *weight* of a data item. More precisely, the "decayed weight" of item (v, w, t, id) at time $c \ge t$ is $w \cdot f(c - t)$. An example decay function is the sliding window model [15, 18, 28], where $f(x)$ is defined as follows. For some window size W, if $x \le W$, then $f(x) = 1$; otherwise, $f(x) = 0$. Other popular decay functions include exponential decay $f(x) = \exp(-ax)$ for a constant a, and polynomial decay, $f(x) = x^{-a}$. An *integral decay function* is one where the decayed weight $w \cdot f(c - t)$ is always an integer. Note that sliding window is trivially such a function; other functions can be made integral by appropriate rounding or scaling.

1.2 Aggregates

Let $f(\cdot)$ denote a decay function, and c denote the time at which a query is posed. Let the set of *distinct* observations in R be denoted by D. We now describe the aggregate functions considered:

1. The *decayed sum* at time c is defined as

$$V = \sum_{(v,w,t,id) \in D} w f(c - t),$$

i.e. the sum of the decayed weights of all distinct elements in the stream. For example, suppose every sensor published one temperature reading every minute, and we are interested in estimating the mean temperature over all readings published in the last 90 minutes. This can be estimated as the ratio of the sum of observed temperatures in the last 90 minutes, to the number of observations in the last 90 minutes. For estimating the sum of temperatures, we consider a data stream where the weight w_i is equal to the observed temperature, and the sum is estimated using a sliding window decay function of 90 minutes duration. For the number of observations, we consider a data stream where for each temperature observation, there is an element where the weight equals to 1, and the decayed sum is estimated over a sliding window of 90 minutes duration.

2. Informally, the *decayed ϕ-quantile* at time c is a value ν such that the total decayed weight of all elements in D whose value is less than or equal to ν is a ϕ fraction of the total decayed weight. For example, in the setting where sensors publish temperatures, each observation may have a "confidence level" associated with it, which is assigned by the sensor. The user may be interested in the weighted median of the temperature observations, where the weight is initially the "confidence level" and decays with time. This can be achieved by setting the value v equal to the observed temperature, the initial weight w equal to the confidence level, $\phi = 0.5$, and using an appropriate time decay function.

   Since computation of exact quantiles (even in the unweighted case) in one pass provably takes space linear in the size of the set [24], we consider approximate quantiles. Our definition below is suited for the case

when the values are integers, and where there could be multiple elements with the same value in D. Let the relative rank of a value u in D at time c be defined as $\frac{\sum_{\{(v,w,t,id)\in D | v \le u\}} wf(c-t)}{\sum_{(v,w,t,id)\in D} wf(c-t)}$. For a user defined $0 < \epsilon < \phi$, the ϵ-approximate decayed ϕ-quantile is a value ν such that the relative rank of ν is at least $\phi - \epsilon$ and the relative rank of $\nu - 1$ is less than $\phi + \epsilon$.

3. *Frequent items.* Let the (weighted) relative frequency of occurrence of value u at time c be defined as

$$\psi(u) = \frac{\sum_{\{(v,w,t,id)\in D | v=u\}} wf(c-t)}{\sum_{(v,w,t,id)\in D} wf(c-t)}.$$

The frequent items are those values ν such that $\psi(\nu) > \phi$ for some threshold ϕ, say $\phi = 2\%$. The exact version of the frequent elements problems requires the frequency of all items to be tracked precisely, which is provably expensive to do in small space [4]. Thus we consider the ϵ-approximate frequent elements problem, which requires us to return all values ν such that $\psi(\nu) > \phi$ and no value ν' such that $\psi(\nu') < \phi - \epsilon$.

4. A *selectivity estimation* query is, given a *predicate* $P(v, w)$ which returns either 0 or 1, to evaluate Q defined as:

$$Q = \frac{\sum_{(v,w,t,id)\in D} wP(v,w)f(c-t)}{\sum_{(v,w,t,id)\in D} wf(c-t)}.$$

Informally, the selectivity of a predicate $P(v, w)$ is the ratio of the total (decayed) weight of all stream elements that satisfy predicate P to the total decayed weight of all elements. Note that $0 \le Q \le 1$. The ϵ-approximate selectivity estimation problem is to return a value $\hat{Q}$ such that $|\hat{Q} - Q| \le \epsilon$.

An exact computation of the duplicate insensitive decayed sum over a general integral decay function is impossible in small space, even in a non-distributed setting. If we can exactly compute a duplicate sensitive sum, we can insert an element e, and test whether the sum changes. The answer determines whether e has been observed already. Since it is possible to reconstruct all the (distinct) elements observed in the stream so far, such a sketch needs space linear in the size of the input, in the worst case. This linear space lower bound holds even for a sketch which can give exact answers with a δ error probability for $\delta < 1/2$ [3], and for a sketch that can give a deterministic approximation [3, 21]; such lower bounds for deterministic approximations also hold for quantiles and frequent elements in the duplicate insensitive model. Thus we look for randomized approximations of all these aggregates; as a result, all of our guarantees are of the form "With probability at least $1 - \delta$, the estimate is an ϵ-approximation to the desired aggregate".

The main contribution of this paper is a general purpose sketch that can estimate all the above aggregates in a general model of sensor data aggregation—with duplicates, asynchronous arrivals, general decay functions, and distributed computation. The space taken by our sketch is logarithmic in the size of the input data, logarithmic in $1/\delta$ where δ is the error probability, and quadratic in $1/\epsilon$, where ϵ is the relative error. There are lower bounds [19] showing that this quadratic dependence on $1/\epsilon$ is necessary for duplicate insensitive computations, showing that our upper bounds are close to optimal.

Outline of the Paper.

After describing related work in Section 2, we consider the construction of a sketch for the case of integral decay functions in Section 3. Although such functions seem initially limiting, they turn out to be the key to solving all possible decay functions efficiently. We observe that the sliding window case can be solved by our algorithm, and more strongly, that the same data structure can answer queries over any sliding window with any window size over the stream. In Section 4, we show a reduction from an arbitrary decay function to the combination of multiple sliding window queries, and demonstrate how this reduction can be performed efficiently; combining these pieces shows the main result of our paper, that arbitrary decay functions can be applied to asynchronous data streams to compute aggregates such as decayed sums, quantiles, frequent elements (or "heavy hitters"), and other related aggregates. A single data structure suffices, and it turns out that even the decay function does not have to be fixed, but can be chosen at evaluation time. We make some concluding observations in Section 5.

2. RELATED WORK

There is a large body of work on data aggregation algorithms in the areas of data stream processing [25] and sensor networks [20, 2, 10]. In this section, we survey algorithms that achieve some of our goals: duplicate insensitivity, time-decaying computations, and asynchronous arrivals in a distributed context — we know of no prior work which achieves all of these simultaneously.

The Flajolet-Martin (FM) sketch [16] is a simple technique to approximately count the number of distinct items observed, and hence is duplicate insensitive. Building on this, Nath, Gibbons, Seshan and Anderson [26] proposed a set of rules to verify whether the sketch is duplicate-insensitive, and gave examples of such sketches. They showed two techniques: FM sketches to compute the COUNT of distinct observations in the sensor network, and a variation of min-wise hashing [7] to draw a uniform, unweighted sample of observed items. Also leveraging on the FM sketch [16], Considine, Li, Kollios and Byers [12] proposed a technique to accelerate multiple updates, and hence yield a duplicate insensitive sketch for the COUNT and SUM aggregates. However, these sketches do not provide a way for the weight of data to decay with time. Once an element is inserted into the sketch, it will stay there forever, with the same weight as when it was inserted into the sketch; it is not possible to use these sketches to compute aggregates on recent observations. Further, their sketches are based on the assumption of hash functions returning values that are completely independent, while our algorithms work with the pairwise independent hash functions. The results of Cormode and Muthukrishnan [13] show duplicate insensitive computations of quantiles, heavy hitters, and frequency moments. They do not consider the time dimension either.

Datar, Gionis, Indyk and Motwani [15] considered how to approximate the count over a sliding window of elements in a data stream under a synchronous arrival model. They presented an algorithm based on a novel data structure called *exponential histogram* for basic counting, and also the reductions from other aggregates, such as sum and ℓ_p norms, to use this data structure. Gibbons and Tirthapura [18] gave an algorithm for basic counting based on a data structure called *wave* with improved worst-case performance. These

algorithms rely explicitly on synchronous arrivals: they partition the input into buckets of precise sizes (typically, powers of two). So it is not clear how to extend to asynchronous arrivals, which would fall into an already "full" bucket. Arasu and Manku [4] presented algorithms to approximate frequency counts and quantiles over a sliding window. The bounds for frequency counts were recently improved by Lee and Ting [22]. Babcock, Datar, Motwani and O'Callaghan [6] presented algorithms for maintaining the variance and k-medians of elements within a sliding window. All of these algorithms rely critically on structural properties of the aggregate being approximated, and use similar "bucketing" approaches to the above methods for counts, meaning that asynchronous arrivals cannot be accommodated. Equally, in all these works, the question of duplicate-insensitivity is not considered.

Going beyond the question of sliding windows, Cohen and Strauss [11] formalized the problem of maintaining *time-decaying* aggregates, and gave strong motivating examples where other decay functions are needed. They demonstrated that any general time-decay function based SUM can be reduced to the sliding window decay based SUM. In this paper, we extend this reduction and show how our data structure supports it efficiently; we also extend the reduction to general aggregates such as frequency counts and quantiles, while guaranteeing duplicate-insensitivity and handling asynchronous arrivals. This arises since we study duplicate-insensitive computations (not a consideration in [11]): performing an approximate duplicate-insensitive count (even without time decay) requires randomization in order to achieve sublinear space [3].

Babcock, Datar and Motwani [5] gave simple algorithms for drawing a uniform sample from a sliding window. To draw a sample of expected size s they keep a data structure of size $O(s \log n)$, where n is the number of items which fall in the window. Recently, Aggarwal [1] proposed an algorithm to maintain a set of sampled elements so that the probability of the rth most recent element being included in the set is (approximately) proportional to $\exp(-ar)$ for a chosen parameter a. An open problem from [1] is to be able to draw samples with an arbitrary decay function, in particular, ones where the timestamps can be arbitrary, rather than implicit from the order of arrival. We partially resolve this question: for an integral decay function, our algorithm maintains a sample such that the probability of retaining any item in the sample is proportional to its current decayed weight and, with high probability, this sample is large enough to accurately estimate useful statistics.

Gibbons and Tirthapura [17] introduced a model of distributed computation over data streams. Each of many distributed parties only observes a local stream and maintains a space-efficient sketch locally. The sketches can be merged by a central site to estimate an aggregate over the union of the streams: in [17], they considered the estimation of the size of the union of distributed streams, or equivalently, the number of distinct elements in the streams. This algorithm was generalized by Pavan and Tirthapura [27] to compute the duplicate-insensitive sum as well as other aggregates such as max-dominance norm. Tirthapura, Xu and Busch [28] proposed the concept of asynchronous streams and gave a randomized algorithm to approximate the sum and median over a sliding window. Here, we extend this line of work to handle both general decay and duplicate arrivals.

3. AGGREGATES OVER AN INTEGRAL DECAY FUNCTION

In this section, we consider a sketch for duplicate insensitive decayed aggregation over a time decay function $f()$, such that the decayed weight $w \cdot f(c-t)$ is always an integer. We first describe the intuition behind our sketch.

3.1 High-level description

Recall that R denotes the observed stream and D denotes the set of distinct elements in R. Though our sketch can provide estimates of many aggregates, for the intuition, we suppose that the task was to answer a query for the decayed sum of elements in D at time κ, i.e.

$$V = \sum_{(v,w,t,id) \in D} w f(\kappa - t).$$

Let w_{max} denote $f(0)$ times the maximum weight of any element and id_{max} the maximum value of id. Consider the following hypothetical process, which happens at query time κ. This process description is for intuition only, and is not executed by the algorithm. For each distinct stream element $e = (v, w, t, id)$, a range of integers is defined as

$$r_e^\kappa = [w_{max} \cdot id, w_{max} \cdot id + wf(\kappa - t) - 1].$$

Note that the size of the range is exactly $wf(\kappa - t)$. Further, if the same element e appears again in the stream, an identical range is defined, and for elements with distinct values of id, the defined ranges are disjoint. Thus we have the following observation.

OBSERVATION 3.1.

$$\sum_{e=(v,w,t,id) \in D} wf(\kappa - t) = \left| \bigcup_{e \in R} r_e^\kappa \right|$$

The integers in r_e^κ are placed in random samples $T_0, T_1, \ldots, T_M$ as follows, where M is of the order of $\log(w_{max} \cdot id_{max})$, which will be precisely defined later. Each integer in r_e^κ is placed in sample T_0. For $i = 0 \ldots M-1$, each integer in T_i is placed in T_{i+1} with probability approximately $1/2$ (the probability is not exactly $1/2$ due to the nature of the sampling functions, which will be made precise later). The probability that an integer is placed in T_i is $p_i \approx 1/2^i$. Then the decayed sum V can be estimated using T_i as the number of integers selected into T_i, multiplied by $1/p_i$. Note that the expected value of an estimate using T_i is V for every i, and by choosing a "small enough" i, we can get an estimate for V that is close to its expectation with high probability.

We now discuss how our algorithm simulates the behavior of the above process under severe space constraints and under online arrival of stream elements. Overcounting due to duplicates is avoided through sampling based on a hash function h, which will be precisely defined later. If an element e appears again in the stream, then the same set of integers r_e^κ is defined (as described above), and the hash function leads to exactly the same decision as before about whether or not to place each integer in T_i. Thus, if an element appears multiple times it is either selected into the sample every time (in which case duplicates are detected and discarded) or it is never selected into the sample.

Another issue is that for an element $e = (v, w, t, id)$, the length of the defined range r_e^κ is $wf(\kappa - t)$, which can be very large. Separately sampling each of the integers in r_e^κ

would require evaluating the hash function $wf(\kappa - t)$ times for each sample, which can be very expensive time-wise, and exponential in the size of the input. So, we make use of the time-efficient *Range-Sampling* technique, introduced in [27] to sample the whole range r_e^κ quickly in time $O(\log |r_e^\kappa|)$, through taking advantage of the structure present in the pairwise independent hash function h. Further, all integers in r_e^κ that have been sampled into T_i are stored together (implicitly) by simply storing an element e in T_i whenever at least one integer in r_e^κ is selected into T_i.

Even with the above Range Sampling technique it is impossible for the algorithm to compute and store the samples T_i in exactly the manner described above. First of all, the query time κ, and hence the weight of an observation, $wf(\kappa - t)$, is unknown at the time the element arrives in the stream. To overcome this problem, we note that the weight at time c is $wf(c-t)$, which is a strictly non-increasing function of c. It is possible to identify an "expiry time" for e at level i, $\text{expiry}(e,i)$ such that as long as $c < \text{expiry}(e,i)$ the range r_e^c has at least one integer selected into T_i, and for $c \geq \text{expiry}(e,i)$, r_e^c has no integers selected into T_i. This way, we can tag e with its expiry time when it arrives, and retain it in T_i only as long as the current time is less than $\text{expiry}(e,i)$. For any future query arriving at time $\kappa \geq \text{expiry}(e,i)$, an estimate for the sum using T_i will never use e.

Next, for smaller values of i, T_i may be too large, and hence take too much space. Here the algorithm stores only the subset S_i of at most τ elements of T_i with the largest expiry times, and discards the rest (τ is a parameter that depends on the desired accuracy). Note that the τ largest elements of any stream can be easily maintained incrementally in one pass through the stream with $O(\tau)$ space. Let the samples actually maintained by the algorithm be denoted $S_0, S_1, \ldots, S_M$.

Now we show an example of computing the time decayed sum in Figure 1. Since the "value" field v is not used, we simplify the element as (w,t,id). The input stream $e_1, e_2, \ldots, e_8$ is shown at the top of the figure. We assume a time decay function where the decayed weight of element (w_i, t_i, id_i) at time t is $\omega_i^t = \lfloor \frac{w}{t-t_i} \rfloor$. Recall that the expiry time of e_i at level j, denoted by $expiry(e_i, j)$, is the earliest time t at which no integers in $r_{e_i}^t$ are selected into level j. The figure only showes the expiry time at level 0. Suppose the current time $c = 15$.

The current state of the sketch is shown in the figure. At the current time, e_1 and e_3 have expired at level 0, which implies they also have expired at all other levels. e_7 and e_8 do not appear in the sketch, because they are duplicates of e_4 and e_5 respectively. Among the remaining elements e_2, e_4, e_5, e_6, only the $\tau = 3$ elements with the largest expiry times are retained in S_0; thus e_4 is discarded from S_0. From the set $\{e_2, e_4, e_5, e_6\}$, a subset $\{e_4, e_5, e_6\}$ is (randomly) selected into S_1 based on the hash values of integers in $r_{e_i}^{15}$ (this implies $expiry(e_4, 1) > 15$, $expiry(e_5, 1) > 15$, $expiry(e_6, 1) > 15$ and $expiry(e_2, 1) \leq 15$), and since there is enough room, all these are stored in S_1. Only e_5 is selected into S_2 and no element is selected into level 3.

When a query is posed for the sum at time 15, the algorithm finds the smallest number ℓ such that the sample S_ℓ has not discarded any element whose expiry time is greater than 15. For example, in Figure 1, $\ell = 1$. Note that at this level, $S_\ell = T_\ell$, and so S_ℓ can be used to answer the query

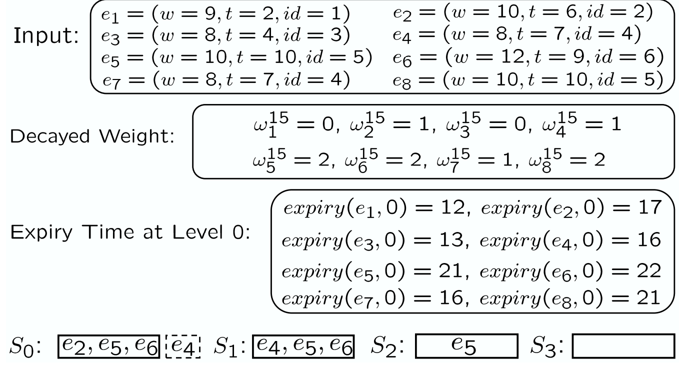

Figure 1: An example stream with 8 elements, and its sketch $\{S_0, S_1, S_2, S_3\}$ for the decayed sum. The current time is 15. The decayed weight of e_i at time t is denoted by ω_i^t. Recall that the expiry time of e_i at level j is denoted by $\text{expiry}(e_i, j)$. The element e_4 in the dashed box indicates that it was discarded from S_0 due to an overflow.

for V. The intuition of choosing such a smallest ℓ is that the expected sample size at level ℓ is the largest among all the samples that can be used to answer the query, and the larger the sample size is, the more accurate the estimate will be. Further, it can be shown with high probability, the estimate for V using S_ℓ is within the desired relative error.

3.2 Formal Description

We now describe how to maintain the different samples $S_0, S_1, \ldots, S_M$. Let h be a pairwise independent hash function chosen from a 2-universal family of hash functions as follows (following Carter and Wegman [9]). Let $\Upsilon = w_{max}(id_{max} + 1)$. The domain of h is $[1 \ldots \Upsilon]$. Choose a prime number p such that $10\Upsilon < p < 20\Upsilon$, and two numbers a and b uniformly at random from $\{0, \ldots, p-1\}$. Then hash function $h : \{1, \ldots, \Upsilon\} \rightarrow \{0, \ldots, p-1\}$ is defined as $h(x) = (a \cdot x + b) \bmod p$. The function $\text{RangeSample}(r, i)$, defined precisely in [27], takes as its input a range $r \subseteq [1, \Upsilon]$ and a sampling level $i \in [0, M]$, and (quickly) returns the number of points $x \in r$ such that $h(x) \in \{0, \ldots, \lfloor 2^{-i} p \rfloor - 1\}$.

Computation of expiry(e, i).

For any element $e = (v, w, t, id)$ and level $0 \leq i \leq M$, the function $\text{expiry}(e, i)$ returns the time $\bar{t}$ such that for $c < \bar{t}$, $\text{RangeSample}(r_e^c, i) > 0$, and for $c \geq \bar{t}$, $\text{RangeSample}(r_e^c, i) = 0$. Note that

$$r_e^c = [w_{max} \cdot id, w_{max} \cdot id + w \cdot f(c-t) - 1].$$

Since $f(c-t)$ is a non-increasing function of c, the range r_e^c shrinks monotonically as c increases. Since $\text{RangeSample}(r_e^c, i)$ is exactly the number of points from r_e^c that are selected into T_i by the hash function, it is also a non-increasing function of c. If $\lim_{t \to \infty} f(t) = f_{min} > 0$, and $\text{RangeSample}(r_e^\infty, i) > 0$, then we define $\text{expiry}(e, i) = \infty$, i.e. e never expires from level i. Similarly, if $\text{RangeSample}(r_e^0, i) = 0$, then $\text{expiry}(e, i)$ is defined to be 0, and e never belongs to level i. Else, we can compute $\bar{t}$ using a binary search on the range of possible times $[t, t + t_{max}]$ where t_{max} is the smallest t' such that $f(t') = f_{min}$.

Initialization:

1. Randomly choose a hash function h as described in Section 3.2
2. For $0 \leq i \leq M$, $S_i \leftarrow \emptyset$, $t_i \leftarrow -1$.
   // t_i is the maximum expiry time among all the elements discarded so far at level i

When a new item $e = (v, w, t, id)$ arrives:
For $0 \leq i \leq M$

1. If $(e \in S_i)$, then return; *// e is a duplicate.*
2. If $(\text{expiry}(e, i) > \max\{c, t_i\})$
   (a) $S_i \leftarrow S_i \cup \{e\}$
   (b) If $|S_i| > \tau$ then *// overflow*
      i. $t_i \leftarrow \min_{e \in S_i} \text{expiry}(e, i)$
      ii. $S_i \leftarrow S_i \backslash \{e | \text{expiry}(e, i) = t_i\}$

To merge two sketches S, S':
For $0 \leq i \leq M$

1. $S_i \leftarrow S_i \cup S'_i$
2. $t_i \leftarrow \max\{t_i, t'_i\}$
3. While $|S_i| > \tau$ do
   (a) $t_i \leftarrow \min_{e \in S_i} \text{expiry}(e, i)$
   (b) $S_i \leftarrow S_i \backslash \{e | \text{expiry}(e, i) = t_i\}$

Figure 2: General Sketch Algorithm over an Integral Decay Function

If e never expires, we determine this with a single call to the function RangeSample, else we make $O(\log \bar{t}) = O(\log t_{max})$ evaluations of RangeSample. The time complexity of the computation of $\text{expiry}(e, i)$ is therefore $O(\log t_{max} \log w_{max})$, since $O(\log w_{max})$ bounds the time cost of RangeSample. The sketch S for an integral decay function is the set of pairs (S_i, t_i), for $i = 0 \dots M$, where S_i is the sample, and t_i is the largest expiry time of any element discarded from S_i so far.

LEMMA 3.1. *The sample S_i is order insensitive; it is unaffected by permuting the order of arrival of the stream elements. The sample is also duplicate insensitive; if the same element e is observed multiple times, the resulting sample is the same as if it had been observed only once.*

PROOF. Order insensitivity is easy to see since S_i is the set of τ elements in T_i with the largest expiry times, and this is independent of the order in which elements arrive. To prove duplicate insensitivity, we observe that if the same element $e = (v, w, t, id)$ is observed twice, the function $\text{expiry}(e, i)$ yields the same outcome, and hence T_i is unchanged, from which S_i is correctly derived. □

LEMMA 3.2. *Suppose two samples S_i and S'_i were constructed using the same hash function h on two different streams R and R' respectively. Then S_i and S'_i can be merged to give a sample of $R \cup R'$.*

PROOF SKETCH. To merge samples S_i and S'_i from two (potentially overlapping) streams R and R', we observe that the required ith level sample of $R \cup R'$ is a subset of the τ elements with the largest expiry times in $T_i \cup T'{}_i$, after discarding duplicates. This can easily be computed from S_i and S'_i. The formal algorithm is given in Figure 2. □

Since it is easy to merge together the sketches from distributed observers, for simplicity the subsequent discussion is framed from the perspective of a single stream. We note that the sketch resulting from merging S and S' gives the same correctness and accuracy with respect to $R \cup R'$ as did S and S' with respect to R and R' respectively.

LEMMA 3.3 (SPACE AND TIME COMPLEXITY). *The space complexity of the sketch for integral decay is $O(M\tau)$ units, where each unit is an input observation (v, w, t, id). The expected time for each update is $O(\log w (\log \tau + \log t_{max} \log w_{max}))$. Merging two sketches takes time $O(M\tau)$.*

PROOF SKETCH. The space complexity follows from the fact that the sketch consists of $M + 1$ samples, and each sample contains at most τ stream elements. For the time complexity, the sample S_i can be stored in a priority queue ordered by expiry times. To insert a new element e into S_i, it is necessary to compute the expiry time of e as $\text{expiry}(e, i)$ once. This takes time $O(\log w_{max} \log t_{max})$. Note that for each element e, we can compute its expiry time at level i exactly once and store the result for later use. An insertion into S_i may cause an overflow, which will necessitate the discarding of elements with the smallest expiry times. In the worst case, all elements in S_i may have the same expiry time, and may need to be discarded, leading to a cost of $O(\tau + \log w_{max} \log t_{max})$ for S_i, and a worst case time of $O(M(\tau + \log w_{max} \log t_{max}))$ in total. But the amortized cost of an insertion is much smaller and is $O(M(\log \tau + \log w_{max} \log t_{max}))$, since the total number of elements discarded due to overflow is no more than the total number of insertions, and the cost of discarding an element due to overflow can be charged to the cost of a corresponding insertion. The expected number of levels into which the element $e = (v, w, t, id)$ is inserted is not M, but only $O(\log w)$, since the expected value of $\text{RangeSample}(r_e^c, i) = p_i |r_e^c| \approx w/2^i$. Thus the expected amortized time of insertion is $O(\log w (\log \tau + \log t_{max} \log w_{max}))$.

Two sketches can be merged in time $O(M\tau)$ since two priority queues (implemented as max-heaps) of $O(\tau)$ elements each can be merged and the smallest elements discarded in $O(\tau)$ time. □

3.3 Computing Decayed Aggregates Using the Sketch

We now describe how to compute a variety of decayed aggregates using the sketch S. For $i = 0 \dots M$, let $p_i = \frac{\lfloor p 2^{-i} \rfloor}{p}$ denote the sampling probability at level i.

Decayed Sum.

We begin with the decayed sum:

$$V = \sum_{(v,w,t,id) \in D} w f(c - t).$$

For computing the decayed sum, let the maximum size of a sample be $\tau = 60/\epsilon^2$, and the maximum number of levels be $M = \lceil \log w_{max} + \log id_{max} \rceil$.

THEOREM 3.1. *The algorithm in Figure 3 yields an estimator $\hat{V}$ of V such that $\Pr[|\hat{V} - V| < \epsilon V] > \frac{2}{3}$. The time taken to answer a query for the sum is $O(\log M + \frac{1}{\epsilon^2} \log w_{max})$. The expected time for each update is $O(\log w (\log \frac{1}{\epsilon} + \log t_{max} \log w_{max}))$. The space complexity is $O(\frac{1}{\epsilon^2}(\log w_{max} + \log id_{max}))$.*

When queried for the decayed sum at time c:

1. $\ell = \min\{i | 0 \le i \le M, t_i \le c\}$
2. If ℓ does not exist, then the algorithm fails, return.
3. If ℓ exists, then return $\frac{1}{p_\ell} \sum_{e \in S_\ell} \text{RangeSample}(r_e^c, \ell)$

Figure 3: Duplicate Insensitive Sum over a Sliding Window

PROOF. We show the correctness of our algorithm for the sum through a reduction to the range-efficient algorithm for counting distinct elements from [27] (we refer to this algorithm as the PT algorithm, for the initials of the authors of [27]). Suppose a query for the sum was posed at time c. Consider the stream $\mathcal{I} = \{r_e^c | e \in R\}$, which is defined on the weights of the different stream elements when the query is posed. From Observation 3.1, we have $|\cup_{r \in \mathcal{I}} r| = V$.

Consider the processing of the stream $\mathcal{I}$ by the PT algorithm. The algorithm samples the ranges in $\mathcal{I}$ into different levels using hash function h. When asked for an estimate of the size of $\cup_{r \in \mathcal{I}} r$, the PT algorithm uses the smallest level, say ℓ', such that the $|\{e \in D | \text{RangeSample}(r_e^c, \ell') > 0\}| \le \tau$, and returns an estimate $Y = \frac{1}{p_{\ell'}} \sum_{e \in D} \text{RangeSample}(r_e^c, \ell')$. From Theorem 1 in [27], Y satisfies the condition $\Pr[|Y - V| < \epsilon V] > 2/3$ if we choose the sample size $\tau = 60/\epsilon^2$, and number of levels M such that $M > \log V_{max}$ where V_{max} is an upper bound on V. Since $w_{max} id_{max}$ is an upper bound on V (each distinct id can contribute at most w_{max} to the decayed sum), our choice of M satisfies the above condition.

Consider the sample S_ℓ used by the algorithm in Figure 3 to answer a query for the sum. Suppose ℓ exists, then ℓ is the smallest integer such that $t_\ell \le c$. For every $i < \ell$, we have $t_i > c$, implying that S_i has discarded at least one element e such that $\text{RangeSample}(r_e^c, i) > 0$. Thus for level $i < \ell$, it must be true that $|\{e | \text{RangeSample}(r_e^c, i) > 0\}| > \tau$, and similarly for level ℓ, it must be true that $|\{e | \text{RangeSample}(r_e^c, \ell) > 0\}| \le \tau$. Thus, if level ℓ exists, then $\ell = \ell'$, and the estimate returned by our algorithm is exactly Y, and the theorem is proved. If ℓ does not exist, then it must be true that for every level $i, 0 \le i \le M$, $|\{e \in D | \text{RangeSample}(r_e^c, i) > 0\}| > \tau$, and thus the PT algorithm also fails to find an estimate.

For the time complexity of a query, observe that finding the right level ℓ can be done in $O(\log M)$ time by organizing the t_is in a search structure, and once ℓ has been found, the function RangeSample() has to be called on the $O(\tau)$ elements in S_ℓ, which takes a further $O(\log w_{max})$ time per call to RangeSample().

The expected time for each update and the space complexity directly follows from Lemma 3.3. □

We note that typically one wants a guarantee that the failure probability is $\delta \ll \frac{1}{3}$. To give such a guarantee, we can keep $\Theta(\log 1/\delta)$ independent copies of the sketch (based on different hash functions), and take the median of the estimates. A standard Chernoff bounds argument shows that the median estimate is accurate within ϵV with probability at least $1 - \delta$.

Selectivity Estimation.

Recall the definition of selectivity estimation from the Introduction. In order to estimate selectivity, we use our sketch to find a sample S_ℓ such that S_ℓ is the lowest numbered sample that has not discarded any element whose expiry time exceeds c, and evaluate the selectivity of P over elements in S_ℓ. We argue that this accurately estimates the selectivity over the whole stream. The formal algorithm is given in Figure 4. For selectivity estimation, setting $\tau = 492/\epsilon^2$ and the maximum number of levels be $M = \lceil \log w_{max} + \log id_{max} \rceil$, we get:

When queried for the selectivity of P at time c:

1. $\ell = \min\{i | 0 \le i \le M, t_i \le c\}$
2. If ℓ does not exist, then the algorithm fails, return.
3. If ℓ exists, then return
$\frac{\sum_{e=(v,w,t,id) \in S_\ell} \text{RangeSample}(r_e^c, \ell) \cdot P(v,w)}{\sum_{e \in S_\ell} \text{RangeSample}(r_e^c, \ell)}$

Figure 4: Selectivity Estimation over an Integral Decay Function

THEOREM 3.2. *The Algorithm in Figure 4 yields an estimate $\hat{Q}$ of Q such that $\Pr[|\hat{Q} - Q| < \epsilon] > 2/3$. The time taken to answer a query for the selectivity of P is $O(\log M + \frac{1}{\epsilon^2} \log w_{max})$. The expected time for each update is $O(\log w (\log \frac{1}{\epsilon} + \log t_{max} \log w_{max}))$. The space complexity is $O(\frac{1}{\epsilon^2}(\log w_{max} + \log id_{max}))$.*

Due to space constraints, we omit the proof here, and refer the reader to the technical report [14]. As in the case of the decayed sum computation, we can amplify the probability of success to $(1 - \delta)$ by taking the median of $\Theta(\log 1/\delta)$ repetitions of the data structure (based on different hash functions).

THEOREM 3.3. *We can answer the ϵ-approximate ϕ-quantiles and frequent items problems using the same sketch data structure, in time $O(\log M + \frac{1}{\epsilon^2} \log(\frac{w_{max}}{\epsilon^2}))$. The expected time for each update is $O(\log w (\log \frac{1}{\epsilon} + \log t_{max} \log w_{max}))$. The space complexity is $O(\frac{1}{\epsilon^2}(\log w_{max} + \log id_{max}))$.*

PROOF SKETCH. The expected time for each update and the space complexity directly follows from Lemma 3.3. Now we show how to reduce a sequence of problems to instances of selectivity estimation.

- A *rank estimation* query for a value ν asks to estimate the (weighted) fraction of elements whose value v is at most ν. This is encoded by a predicate P_ν such that $P_{\le \nu}(v, w) = 1$ if $v \le \nu$, else 0. Clearly, this can be solved using the above analysis with additive error at most ϵ.

- The *median* is the item whose rank is 0.5. To find the median, we can binary search for the smallest ν such that the rank of ν is 0.5 or higher, using the selectivity of the P_ν predicate. We pose at most $\log v_{max}$ queries in our binary search, and by the union bound, they all succeed with probability at least $(1 - \delta) \log v_{max}$.

- *Quantiles* generalize the median to find items whose ranks are multiples of ϕ, e.g. the quantiles, which are elements at ranks 0.2, 0.4, 0.6 and 0.8. Again, by binary searching for each one in turn, we can find them with additive error ϵ.

- The *frequent items* problem can also be solved using selectivity queries. Value ν is a frequent item iff the selectivity of the predicate "$P_{=\nu}(v, w) = 1$ if $v = \nu$" is ϕ or more. The algorithm for frequent elements can check the selectivity of every value occurring within sample S_ℓ, and check if any of them has a selectivity above the desired threshold.

We note that a faster way to compute the median and quantiles is to compute the required quantiles of an appropriate weighted sample S_ℓ where ℓ is defined (as before) as the smallest integer such that $t_\ell \leq c$. The result of this computation is the same as that found by the method above, but is faster to compute. Thus, after computing the sample level and RangeSample values, we can sort the sample by value and so compute frequent items and quantiles, incurring an additional $O(\tau \log \tau)$ cost over the time complexity of selectivity estimation. $\square$

4. GENERAL DECAY FUNCTIONS VIA SLIDING WINDOW

4.1 Sliding Window Decay

Recall that a sliding window decay function, given a window size W, is defined as $f_W(x) = 1$ if $x < W$, and $f_W(x) = 0$ otherwise. As already observed, the sliding window decay function is a perfect example of an integral decay function, and hence we can use the algorithm from Section 3. It also simplifies the application of the algorithm somewhat: we can easily compute the expiry time of any element e_i as $t_i + W$. We can prove a stronger result though: If we set $f(x) = 1$ always when inserting the element. i.e., the element never expires, and discard the element with the oldest timestamp when the sample is full, we can keep a single data structure that is good for *any* sliding window size $W < \infty$, where any W can be specified after the data structure has been created, to return a good estimate of the aggregates.

THEOREM 4.1. *Our data structure can answer sliding window sum and selectivity queries where the parameter W is provided at query time. Precisely, for $\tau = O(\frac{1}{\epsilon^2})$ and $M = O(\log w_{max} + \log id_{max})$, we can provide an estimate $\hat{V}$ of the decayed sum V such that $\Pr[|\hat{V} - V| < \epsilon V] > \frac{2}{3}$ and an estimate $\hat{Q}$ of the selectivity, Q, such that $\Pr[|\hat{Q} - Q| < \epsilon] > \frac{2}{3}$. The time to answer both queries is $O(\log M + \tau)$.*

PROOF. Observe that for all parameters W, the expiry *order* is the same: e_j expires before e_k if and only if $t_j < t_k$. So we keep the data structure as usual, but instead of aggressively expiring items, we keep the τ most recent items at each level i as S_i. Let t_i denote the largest timestamp of the discarded items from level i. We only have to update S_i when a new item arrives in the level. If there are fewer than τ items at the level, we retain it. Else, we either reject the new item if $t \leq t_i$, or else retain it, eject the oldest item in the S_i, and update t_i accordingly. For both sum and selectivity estimation, we find the lowest level where no elements which fall within the window have expired—this is equivalent to the level ℓ from before. From this level, we can extract the sample of items which fall within the window, which are exactly the set we would have if we had enforced the expiry times. Hence, we obtain the guarantees that follow from Theorems 3.1 and 3.2. The running time is slightly faster now: since we compute the expiry time $t + W$ once and for all, at insertion time we can also compute and store the value of the RangeSample predicate run on the inserted range, and so do not need to recompute this at query time. $\square$

Similarly, we can amplify the probability of success to $1 - \delta$ by taking the median of $\Theta(1/\delta)$ repetitions of the data structures, each of which is based on different hash functions.

4.2 Reduction from General Decay Function to Sliding Window Decay

We now give a reduction from the problem of computing the decayed sum using a general time decay function to the problem of computing the sum over a sliding window. The randomized reduction generalizes a (deterministic) idea from Cohen and Strauss [11]: rewrite the decayed computation as the combination of many sliding window queries (over different sized windows). We further show how this reduction can be done in a time-efficient manner.

Selectivity Estimation.

LEMMA 4.1. *Arbitrary decay function selectivities estimation can be rewritten as the combinations of at most $2c$ sliding window queries, where c is the current time value.*

PROOF SKETCH. Let the set of distinct observations in the stream (now sorted by timestamps) be $D = \langle e_1 = (v_1, w_1, t_1, id_1), e_2 = (v_2, w_2, t_2, id_2), \ldots, e_n = (v_n, w_n, t_n, id_n)\rangle$. The decayed selectivity of P at time c using a decay function f is defined as

$$Q = \sum_{(v,w,t,id)\in D} w \cdot P(v, w) \cdot f(c-t) / \sum_{(v,w,t,id)\in D} w \cdot f(c-t), \quad (1)$$

and can be rewritten as $Q = A/B$ where,

$$\begin{aligned} A &= f(c-t_1)\sum_{i=1}^{n} w_i P(v_i, w_i) \\ &+ \sum_{t=t_1+1}^{t_n} \left([f(c-t) - f(c-t+1)] \cdot \sum_{\{i|t_i \geq t\}} P(v_i, w_i) w_i \right) \end{aligned}$$

$$\begin{aligned} B &= f(c-t_1)\sum_{i=1}^{n} w_i \\ &+ \sum_{t=t_1+1}^{t_n} \left([f(c-t) - f(c-t+1)] \cdot \sum_{\{i|t_i \geq t\}} w_i \right) \end{aligned}$$

We compute A and B separately; first, consider B, which is equivalent to V, the decayed sum under the function f. Write V^W for the decayed sum under the sliding window of size W. We can compute $\hat{V} = \sum_{t=t_1+1}^{t_n} \left([f(c-t) - f(c-t+1)] \cdot V^{c-t}\right)$, using the sliding window algorithm for the sum to estimate each V^{c-t}, from $t = t_1 + 1$ till t_n. We also add $(\sum_i w_i) f(c - t_1)$, by tracking $\sum_i w_i$ exactly. Applying our algorithm, each sliding window query V^W is accurate up to a $(1 \pm \epsilon)$ relative error with probability at least $1 - \delta$, so taking the sum of $(t_n - t_1) \leq c$ such queries yields an answer that is accurate with a $(1 \pm \epsilon)$ factor with probability at least $1 - c\delta$, by the

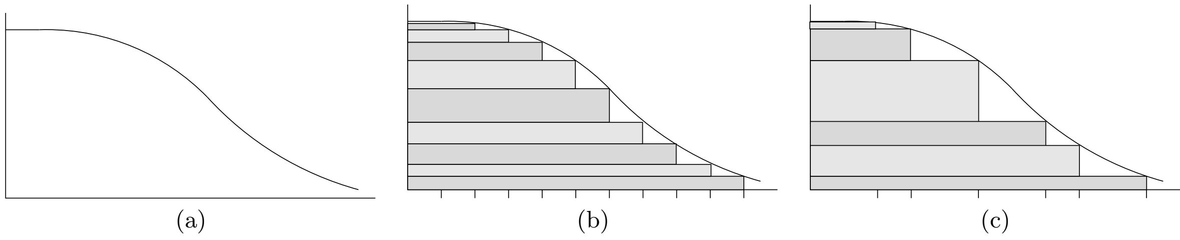

Figure 5: Reduction of general decay functions to sliding window: (a) a sample decay function (b) breaking the decay function into sliding windows every timestep (c) computing sliding windows only for stored timestamps.

union bound. Similarly, A can also be computed by using the sliding window algorithm for the sum. Further, Theorem 4.1 implies each sliding window query in A is accurate up to a $(\pm\epsilon V)$ additive error with probability at least $1-\delta$, which further yields an estimate for A with the accuracy up to $(\pm\epsilon V)$ additive error with probability at least $1-c\delta$. Combining the estimates for A and B and using $\tau = O(1/\epsilon^2)$, we get $|Q'-Q| \leq \epsilon$ with probability at least $(1-2c\delta)$, where Q' is the estimate of A/B. To give the required overall probability guarantee, we can adjust δ by a factor of $2c$. Since the total space and time taken depend only logarithmically on $1/\delta$, scaling δ by a factor of $2c$ increases the space and time costs by a factor of $O(\log c)$. □

THEOREM 4.2. *We can answer selectivity queries with arbitrary decay functions in time $O(M\tau \log(\frac{M\tau}{\delta}) \log(M\tau \log \frac{M\tau}{\delta}))$ to find $\hat{Q}$ so that $\Pr[|Q - \hat{Q}| > \epsilon] < \delta$.*

PROOF. Implementing the above reduction directly would be too slow, depending linearly on the range of timestamps. However, we can improve this by making some observations on the specifics of our implementation of the sliding window sum. Observe that since our algorithm stores at most τ timestamps at each of M levels. So if we probe it with two timestamps $t_j < t_k$ such that, over all timestamps stored in the S_i samples, there is no timestamp t such that $t_j < t \leq t_k$, then we will get the same answer for both queries. Let t_i^j denote the jth timestamp in ascending order in S_i. We can compute the exact same value for our estimate of (1) by only probing at these timestamps, as:

$$\sum_{i=0}^{M} \sum_{\substack{j=1 \\ t_i^j < t_{i-1}^{min}}}^{|S_i|} (f(c - t_i^j) - f(c - t_i^{j+1}))V^{c - t_i^j} \quad (2)$$

where for $0 \leq i \leq M$, t_i^{min} denotes the smallest (oldest) timestamp of the items in S_i, and $t_{-1}^{min} = c + 1$, where c is the current time (this avoids some double counting issues). This process is illustrated in Figure 5: we show the original decay function, and estimation at all timestamps and only a subset. The shaded boxes denote window queries: the length is the size, W of the query, and the height gives the value of $f(c - t_i^j) - f(c - t_i^{j+1})$.

We need to keep $b = \log \frac{M\tau}{\delta}$ independent copies of the data structure (based on different hash functions) to give the required accuracy guarantees. We answer a query by taking the median of the estimates from each copy. Thus, we can generate the answer by collecting the set of timestamps from all b structures, and working through them in sorted order of recency. In each structure we can incrementally work through level by level: for each subsequent timestamp, we modify the answer from the structure that this timestamp originally came from (all other answers stay the same). We can track the median of the answers in time $O(\log b)$: we keep the b answers in sorted order, and one changes each step, which can be maintained by standard dictionary data structures in time $O(\log b)$. If we exhaust a level in any structure, then we move to the next level and find the appropriate place based on the current timestamp. In this way, we work through each data structure in a single linear pass, and spend time $O(\log b)$ for every time step we pass. Overall, we have to collect and sort $O(M\tau b)$ timestamps, and perform $O(M\tau b)$ probes, so the total time required is bounded by $O(M\tau b \log(M\tau b))$. This yields the bounds stated above. □

Once selectivity can be estimated, we can use the same reductions as in the sliding window case to compute time decayed ranks, quantiles, and frequent items, yielding the same bounds for those problems.

Decayed Sum Computation.

We observe that the maintenance of the decayed sum over general decay functions has already been handled as a subproblem within selectivity estimation.

LEMMA 4.2. *Arbitrary decay function sum can be rewritten as the combinations of at most c sliding window queries, where c is the current time.*

THEOREM 4.3. *We can answer arbitrary decay function sum estimation in time $O(M\tau \log(\frac{M\tau}{\delta}) \log(M\tau \log(\frac{M\tau}{\delta})))$ to find $\hat{V}$ such that $\Pr[|\hat{V} - V| > \epsilon V] < \delta]$.*

5. CONCLUDING REMARKS

We observe that this is a very powerful result: not only does there exist a data structure allowing the duplicate-insensitive, distributed computation of aggregates over asynchronous data streams, but also, via the reduction to sliding window computations, there exists a single data structure which can answer decayed aggregate queries without knowledge of the desired decay function until the query is posed.

Given an integral decay function, we have a choice: either compute it directly via the method in Section 3, or compute it via a reduction to sliding windows, using the method in Section 4. There are pros and cons to each approach: the method of Section 3 is faster to evaluate queries on, since we can immediately calculate the result from the appropriate level, whereas the sliding window reduction requires a linear time pass through the data structure. The space used is slightly smaller with the former method, as we need slightly

fewer repetitions to give the necessary probability guarantees. Meanwhile, the method of Section 4 is much more general, allowing us to specify the decay function after the sketch has been made.

An open question is therefore if there is more power in fixing a decay function *a priori*: can we create a smaller sketch (and hence reduce communication when sending it) for certain fixed functions? For certain classes, such as exponential decay functions, which seem to be easier to handle for other aggregates, it seems possible, but no results are known that are better than the general case.

6. REFERENCES

[1] C. C. Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *VLDB*, 2006.

[2] I. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci. A survey on sensor networks. *IEEE Commun. Mag. 40 (8) (2002) 102–114*

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

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

[5] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA*, 2002.

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

[7] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *STOC*, 1998.

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

[9] J.L. Carter and M.L. Wegman. Universal classes of hash functions. *J. of Comp. and System Sciences*, 18(2):143–154, 1979.

[10] Y. Chen, H. V. Leong, M. Xu, Jiannong Cao, K.C.C Chan and A. T.S Chan. In-network data processing for wireless sensor networks. In *MDM (International Conference on Mobile Data Management)* 2006

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

[12] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, 2004.

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

[14] G. Cormode, S. Tirthapura, B. Xu. Time-Decaying Sketches for Sensor Data Aggregation. Technical Report TR-2007-06-0, Department of Electrical and Computer Engineering, Iowa State University.

[15] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. on Computing*, 31(6):1794–1813, 2002.

[16] P. Flajolet and G. N. Martin. Probabilistic counting. In *FOCS*, 1983.

[17] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *SPAA*, 2001.

[18] P. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *SPAA*, 2002.

[19] P. Indyk, D. Woodruff. Tight lower bounds for the distinct elements problem. In *FOCS*, 2003.

[20] N. Kimura and S. Latifi. A survey on data compression in wireless sensor networks. In *ITCC International Conference on Information Technology Coding and Computing*, 2005

[21] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

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

[23] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Operating Systems Review*, 36(SI):131–146, 2002.

[24] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12(3):315–323, 1980.

[25] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Foundations and Trends in Theoretical Computer Science. Now Publishers, August 2005.

[26] S. Nath, P. B. Gibbons, S. Seshan, and Z. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SENSYS*, 2004.

[27] A. Pavan and S. Tirthapura. Range-efficient computation of F_0 over massive data streams. In *SIAM Journal on Computing*, 37(2):359–379, 2007.

[28] S. Tirthapura, B. Xu, and C. Busch. Sketching asynchronous streams over a sliding window. In *PODC*, 2006.