TIME-DECAYING SKETCHES FOR ROBUST AGGREGATION OF SENSOR DATA [*]

GRAHAM CORMODE[†], SRIKANTA TIRTHAPURA[‡], AND BOJIAN XU[§]

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 [32, 15], 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 loss of accuracy. To our knowledge, this is the first sketch that combines all the above properties.

Key words. sensor network, data streams, time decay, asynchrony, data aggregation, duplicates

AMS subject classifications. 68P15, 68P05, 68W20, 68W25, 68W40

1. Introduction.

The growing size and scope of sensor networks has led to greater demand for energy-efficient communication of sensor data. Although sensors are increasing in computing ability, they remain constrained by the cost of communication, since this is the primary drain on their limited battery power. It is widely agreed that the working life of a sensor network can be extended by algorithms which limit communication [29]. In particular, this means that although sensors may observe large quantities of information over time, they should preferably 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

[*]A preliminary version of this paper appeared in the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC) 2007, pages 215-224

[†]AT&T Labs–Research, 180 Park Avenue Florham Park, NJ-07932.(graham@research.att.com)

[‡]Department of Electrical and Computer Engineering, Iowa State University, Ames, IA-50011. (snt@iastate.edu). This author's research was supported in part by NSF grant CNS-0520102 and by ICUBE, Iowa State University

[§]Department of Electrical and Computer Engineering, Iowa State University, Ames, IA-50011. (bojianxu@iastate.edu). This author's research was supported in part by NSF grant CNS-0520102 and by ICUBE, Iowa State University 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) [32, 15]. 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 an arbitrary function that assigns a weight to each observation as a function of its initial weight and its age [18, 14]. 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 by a single sketch which is based on a hash-based sampling procedure that allows a variety of aggregates to be computed efficiently under a general class of 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, $1 \leq i \leq n$ 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 [35, 10]) has focused on the case of *synchronous* streams, where the elements of the stream are assumed to arrive in the order of timestamps. There is a growing body of work (for example, [7]) that recognizes that the assumption of synchronous streams may not always be practical and that there is a need to study more complex temporal models, such as asynchronous arrivals.

Decay Functions. The *age* of an element is defined as the elapsed time since the element was created. Thus, the age of element (v, w, t, id) at time c is $c - t$. A decay function takes the initial weight and the age of an element and returns its *decayed weight.*

DEFINITION 1.1. *A decay function $f(w, x)$ takes two parameters, the weight $w \geq 0$, and an integral age $x \geq 0$, and should satisfy the following conditions. (1) $f(w, x) \geq 0$ for all w, x; (2) if $w_1 > w_2$, then $f(w_1, x) \geq f(w_2, x)$; (3) if $x_1 > x_2$, then $f(w, x_1) \leq f(w, x_2)$.*

The decayed weight of an element (v, w, t, id) at time $c \geq t$ is $f(w, c - t)$. An example decay function is the sliding window model [18, 22, 35], where $f(w, x)$ is defined as follows. For some window size W, if $x \leq W$, then $f(w, x) = w$; otherwise, $f(w, x) = 0$. Other popular decay functions include exponential decay $f(w, x) = w \cdot \exp(-ax)$ and polynomial decay, $f(w, x) = w \cdot (x + 1)^{-a}$, where a is a constant.

DEFINITION 1.2. *A decay function $f(w, x)$ is an* integral decay function *if $f(w, x)$ is always an integer.*

For example, sliding window decay is trivially such a function. Another integral decay function is: $f(w, x) = \left\lfloor \frac{w}{2^x} \right\rfloor$. The class of *decomposable* decay functions is defined as follows.

DEFINITION 1.3. *A decay function $f(w, x)$ is a decomposable decay function if it can be written in the form $f(w, x) = w \cdot g(x)$ for some function $g()$.*

Note that the conditions on a decay function $f(w, x)$ naturally impose the following conditions on $g()$: (1) $g(x) \geq 0$ for all x; (2) if $x_1 < x_2$, then $g(x_1) \geq g(x_2)$. To our knowledge, all previous work on computing time decayed aggregates on streams, including [14, 6, 18, 23, 19, 4, 35, 10, 28, 36] considered decomposable decay functions. For example, exponential decay, sliding window decay, and polynomial decay are all decomposable.

1.2. Aggregates. Let $f(\cdot, \cdot)$ 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:

Decayed Sum. At time c the decayed sum is defined as

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

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 or two, 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.

Decayed ϕ-Quantile. 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 [30], 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
$\left(\sum_{\{(v,w,t,id) \in D : v \leq u\}} f(w, c - t)\right) / \left(\sum_{(v,w,t,id) \in D} f(w, c - t)\right)$. 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 $\phi + \epsilon$.

Decayed Frequent Items. Let the (weighted) relative frequency of occurrence of value u at time c be defined as

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

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 $\psi(\nu') < \phi - \epsilon$.

Decayed Selectivity Estimation. A *selectivity estimation* query is, given a *predicate* $P(v, w)$ which returns 0 or 1 as a function of v and w, to evaluate Q defined as:

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

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 $0 \leq Q \leq 1$. The ϵ-approximate selectivity estimation problem is to return a value $\hat{Q}$ such that $|\hat{Q} - Q| \leq \epsilon$.

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 this would make it 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, 27]; 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".

1.3. Contribution.

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, broad class of decay functions, and distributed computation. The sketch can accommodate any integral decay function, or any decomposable decay function. As already noted, to our knowledge, the class of decomposable decay functions includes all the decay functions that have been considered in the data stream literature so far. The space complexity of the 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 [24] showing that the quadratic dependence on $1/\epsilon$ is necessary for duplicate insensitive computations on data streams, thus implying that our upper bounds are close to optimal.

Beyond the preliminary version of this paper [17], we have the following new contributions.

1. Our algorithm for an integral decay function is based on random sampling, and this paper proposes a novel technique that can quickly determine the time till which an item must be retained within a sample (this is called as the "expiry time" of the item). This technique may be of independent interest. Given a range of integers, it can quickly return the smallest integer of the range selected by a pairwise independent random sampling (or detect that such an integer does not exist).

2. In an extensive experimental evaluation, we observed that the space required by the sketch in practice can be an order of magnitude smaller than the theoretical predictions, while still meeting the accuracy demands. Further, they confirm that the sketch can be updated quickly in an online fashion, allowing for high throughput data aggregation.

Outline of the Paper. After describing related work in Section 2, we consider the construction of a sketch for the case of integral decay in Section 3. Although such functions initially seem limiting, they turn out to be the key to solving the class of decomposable decay functions efficiently. In Section 4, we show a reduction from an arbitrary decomposable decay function to a combination of multiple sliding window queries, and demonstrate how this reduction can be performed efficiently; combining these pieces shows that arbitrary decomposable 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. In Section 5, we present the results of our experiments. We make some concluding observations in Section 6.

2. Related Work.

There is a large body of work on data aggregation algorithms in the areas of data stream processing [31] and sensor networks [25, 2, 12]. 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 [20] 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 [32] proposed a set of rules to verify whether the sketch is duplicate-insensitive, and gave examples of such sketches. They showed two techniques that obey these rules: FM sketches to compute the COUNT of distinct observations in the sensor network, and a variation of min-wise hashing [9] to draw a uniform, unweighted sample of observed items. Also leveraging the FM sketch [20], Considine, Li, Kollios and Byers [15] 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 [16] show duplicate insensitive computations of quantiles, heavy hitters, and frequency moments. They do not consider the time dimension either.

Datar, Gionis, Indyk and Motwani [18] 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 presented reductions from other aggregates, such as sum and ℓ_p norms, to use this data structure. Gibbons and Tirthapura [22] gave an algorithm for basic counting based on a data structure called *wave* with improved worst-case performance. Subsequently, Braverman and Ostrowsky [8] defined Smooth Histograms, a generalization of exponential histograms that take further advantage of the aggregation function (such as SUM and norm computations) to reduce the space required. 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 space bounds for frequency counts were recently improved by Lee and Ting [28]. 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. In all these works, the question of duplicate-insensitivity is not considered except in Datar, Gionis, Indyk and Motwani [18], Section 7.5, where an approach to count the distinct values in a sliding window is briefly described.

Cohen and Strauss [14] formalized the problem of maintaining *time-decaying* aggregates, and gave strong motivating examples where functions other than sliding windows and exponential decay 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 [14]): performing an approximate duplicate-insensitive count (even without time decay) requires randomization in order to achieve sublinear space [3]. Subsequently, Kopelowitz and Porat [26] showed that the worst-case space of this approach for decayed SUM can be improved by more carefully handling the number of bits used to record timestamps, bucket indices, and so on, reducing the costs by logarithmic factors. They also provided lower bounds for approximations with additive error but did not consider duplicate-insensitive computation. Cohen and Kaplan [13] considered spatially-decaying aggregation over network data, based on tracking lists of identies of other nodes in the network chosen via hash functions.

Our results can be viewed as an algorithm for maintaining a sample from the stream, where the probability of an item being present in the sample is proportional to the current decayed weight of that item. Prior work for sampling with weighted decay includes Babcock, Datar and Motwani [5] who 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, by showing a scheme for the case of integral decay functions.

Gibbons and Tirthapura [21] 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 [21], 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 [33] to compute the duplicate-insensitive sum as well as other aggregates such as max-dominance norm. Xu, Tirthapura, and Busch [35] 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 present a sketch for duplicate insensitive time-decayed aggregation over an integral decay function $f()$. 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 multiple 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.

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

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

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

Note that the size of this range, r_e^κ, is exactly $f(w, \kappa - t)$. 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.

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

The integers in r_e^κ are placed in random samples $T_0, T_1, \ldots, T_M$ as follows. M is of the order of $\log(w_{max} \cdot id_{max})$, and will be precisely defined in Section 3.4. 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 $1/p_i$. It is easy to show 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 space constraints and under online arrival of stream elements. Over counting 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 h 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 $f(w, \kappa - t)$, which can be very large. Separately sampling each of the integers in r_e^κ would require evaluating the hash function $f(w, \kappa - t)$ times for each sample, which can be very expensive time-wise, and exponential in the size of the input. Similarly, storing all the selected integers in r_e^κ could be expensive, space-wise. Thus, we store all the sampled integers in r_e^κ together (implicitly) by simply storing the element e in T_i, as long as there is at least one integer in r_e^κ sampled into T_i. However, the query time κ, and hence the weight of an observation, $f(w, \kappa - t)$, are unknown at the time the element arrives in the stream, which means the range r_e^κ is unknown when e is processed. To overcome this problem, we note that the weight at time κ, $f(w, \kappa - t)$, is a non-increasing function of κ, and hence r_e^κ is a range that shrinks as κ increases. We define the "expiry time" of element e at level i, denoted by $\text{expiry}(e, i)$, as the smallest value of κ such that r_e^κ has no sampled elements in T_i. We store e in T_i as long as the current time is less than $\text{expiry}(e, i)$. For any queries issued at time $\kappa \geq \text{expiry}(e, i)$, there will be no contribution from e to the estimate using level i, and hence e does not have to be stored in T_i. In Section 3.3, we present a fast algorithm to compute $\text{expiry}(e, i)$.

Next, for smaller values of i, T_i may be too large (e.g. T_0 is the whole input seen so far), 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 of derived values 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 $S_0, S_1, \ldots, S_M$.

Upon receiving a query for V at time κ, we can choose the smallest i such that $S_i = T_i$, and use S_i to estimate V. In particular, for each element e in T_i, the time-efficient *Range-Sampling* technique, introduced in [33], can be used to return the number of selected integers in the range r_e^κ quickly in time $O(\log |r_e^\kappa|)$.

We show an example of computing the time decayed sum in Figure 3.1. 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 that the decayed weight of an element (w_i, t_i, id_i) at time t is $\omega_i^t = f(w_i, t - t_i) = \lfloor \frac{w_i}{t - t_i} \rfloor$. The figure only shows the expiry times of elements 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 $\text{expiry}(e_4, 1) > 15$, $\text{expiry}(e_5, 1) > 15$, $\text{expiry}(e_6, 1) > 15$ and $\text{expiry}(e_2, 1) \leq 15$), and since there is enough room, all these are stored in S_1. Only

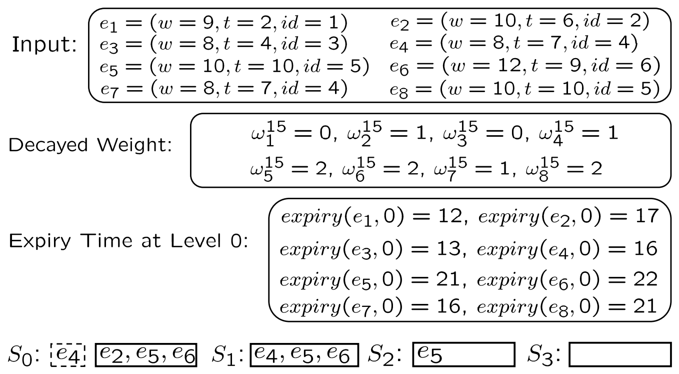

Input:
$e_1 = (w = 9, t = 2, id = 1)$
$e_2 = (w = 10, t = 6, id = 2)$
$e_3 = (w = 8, t = 4, id = 3)$
$e_4 = (w = 8, t = 7, id = 4)$
$e_5 = (w = 10, t = 10, id = 5)$
$e_6 = (w = 12, t = 9, id = 6)$
$e_7 = (w = 8, t = 7, id = 4)$
$e_8 = (w = 10, t = 10, id = 5)$
Decayed Weight:
$\omega_1^{15} = 0,\ \omega_2^{15} = 1,\ \omega_3^{15} = 0,\ \omega_4^{15} = 1$
$\omega_5^{15} = 2,\ \omega_6^{15} = 2,\ \omega_7^{15} = 1,\ \omega_8^{15} = 2$
Expiry Time at Level 0:
$expiry(e_1, 0) = 12,\ expiry(e_2, 0) = 17$
$expiry(e_3, 0) = 13,\ expiry(e_4, 0) = 16$
$expiry(e_5, 0) = 21,\ expiry(e_6, 0) = 22$
$expiry(e_7, 0) = 16,\ expiry(e_8, 0) = 21$
S_0: e_4 e_2, e_5, e_6 S_1: e_4, e_5, e_6 S_2: e_5 S_3:

FIG. 3.1. *An example stream with 8 elements arriving in the order* $e_1, e_2, \ldots, e_8$*, 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*. The expiry time of* e_i *at level* j *is denoted by* $\mathrm{expiry}(e_i, j)$*. The element* e_4 *in the dashed box indicates that it was discarded from* S_0 *due to an overflow caused by more than* $\tau = 3$ *elements being selected into* T_0*.*

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 3.1, $\ell = 1$. Note that at this level, $S_\ell = T_\ell$, and so S_ℓ can be used to answer the query 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 has error that is a function of τ; by choosing τ appropriately, we can ensure that the error is small.

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 [11]). 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 $\{0, \ldots, p-1\}$. The hash function $h : \{1, \ldots, \Upsilon\} \to \{0, \ldots, p-1\}$ is defined as
$h(x) = (a \cdot x + b) \mod p$. We define the expiry time of an element $e = (v, w, t, id)$ at sample level i as follows.

Let $A_e^i = \left\{\bar{t} \geq t : |r_e^{\bar{t}}| > 0 \text{ and for all } x \in r_e^{\bar{t}}, h(x) > \lfloor 2^{-i}p - 1 \rfloor\right\}$. Set A_e^i is the set of clock times at which range $r_e^{\bar{t}}$ is not empty (meaning $f(w, \bar{t} - t) > 0$), but has no integers selected by the hash function h at level i. Note that when $\bar{t}$ becomes larger, range $r_e^{\bar{t}}$ shrinks and eventually becomes empty, so the size of A_e^i is finite and can be 0.

Let $B_e = \left\{\bar{t} \geq t : |r_e^{\bar{t}}| = 0\right\}$. Set B_e is the set of clock times at which range $r_e^{\bar{t}}$ is empty (meaning $f(w, \bar{t} - t) = 0$). We assume that for every decay function f we

Algorithm 1: Initialization(M)

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

Algorithm 2: ProcessItem($e = (v, w, t, id)$)

```
1 for $0 \leq i \leq M$ do
2     if ($e \in S_i$) then return ; // $e$ is a duplicate.
3     if (expiry$(e, i) > \max\{c, t_i\}$) then
4         $S_i \leftarrow S_i \cup \{e\}$;
5         if $|S_i| > \tau$ then // overflow
6             $t_i \leftarrow \min_{e \in S_i}$ expiry$(e, i)$ ;
7             $S_i \leftarrow S_i \backslash \{e : \text{expiry}(e, i) = t_i\}$;
```

consider, there is some finite time t_{max} such that $f(w, t_{max}) = 0$ for every possible weight w, so B must be non-empty.

It is obvious that if $A_e^i \neq \emptyset$, then $\min(A_e^i) < \min(B_e)$ must be true, because $f(w, \bar{t} - t) > 0$ for any $\bar{t} \in A_e^i$, but $f(w, \bar{t} - t) = 0$ for any $\bar{t} \in B_e$, so all the clock times in set A must be smaller than all the clock times in set B.

Definition 3.1. *For stream element $e = (v, w, t, id)$, and level $0 \leq i \leq M$:*

$$\text{expiry}(e, i) = \begin{cases} \min(A_e^i) & \textit{if } A_e^i \neq \emptyset \\ \min(B_e) & \textit{otherwise} \end{cases}$$

Intuitively, expiry(e, i) is the earliest clock time $\bar{t}$, at which either the corresponding non-empty integral range $r_e^{\bar{t}}$ has no integers selected by hash function h at level i or the decayed weight of e becomes 0.

The sketch S for an integral decay function is the set of pairs (S_i, t_i), for $i = 0 \ldots M$, where S_i is the sample, and t_i is the largest expiry time of any element discarded from S_i so far. The formal description of the general sketch algorithm over an integral decay function is shown in Algorithms 1 and 2.

Lemma 3.2. *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 $e = (v, w, t, id)$

Algorithm 3: MergeSketches(S, S')

```
1 for $0 \le i \le M$ do
2     $S_i \leftarrow S_i \cup S'_i$ ;
3     $t_i \leftarrow \max\{t_i, t'_i\}$;
4     while $|S_i| > \tau$ do
5         $t_i \leftarrow \min_{e \in S_i} \text{expiry}(e, i)$ ;
6         $S_i \leftarrow S_i \backslash \{e : \text{expiry}(e, i) = t_i\}$ ;
```

is observed twice, the function expiry(e, i) yields the same outcome, and hence T_i is unchanged, from which S_i is correctly derived. ▯

Lemma 3.3. *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. 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 Algorithm 3. ▯

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.

Theorem 3.4 (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 $O(\log w(\log \tau + \log w + \log t_{max}))$. Merging two sketches takes time $O(M\tau)$.*

Proof. 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 expiry(e, i) once. This takes time $O(\log w + \log t_{max})$ (Section 3.3). 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 $O(\tau + \log w + \log t_{max})$ for S_i, and a worst case time of $O(M(\tau + \log w + \log t_{max}))$ in total. But the amortized cost of an insertion is much smaller and is $O(\log w(\log \tau + \log w + \log t_{max}))$, 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 $|\{h(x) \le \lfloor 2^{-i}p \rfloor : x \in r_e^c\}| = p_i |r_e^c| \approx w/2^i$. Thus the expected amortized time of insertion is $O(\log w(\log \tau + \log w + \log t_{max}))$.

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. Computation of Expiry Time. We now present an algorithm which, given an element $e = (v, w, t, id)$ and level $i, 0 \le i \le M$, computes expiry(e, i). Recall that expiry(e, i) is defined as the smallest integer $\kappa \ge t$ such that either $f(w, \kappa - t) = 0$ (meaning $|r_e^\kappa| = 0$) or $|\{x \in r_e^\kappa : |r_e^\kappa| > 0, h(x) \le \lfloor 2^{-i}p \rfloor\}| = 0$. Let $s_e^i = \min\{x \in r_e^t : h(x) \in \{0, 1, \ldots, \lfloor 2^{-i}p \rfloor - 1\}\}$. Note that s_e^i may not exist. We define Δ_e^i as follows. If s_e^i exists, then $\Delta_e^i = s_e^i - w_{max} \cdot id \ge 0$; else, $\Delta_e^i = -1$. In the following lemma, we show that given Δ_e^i, it is easy to compute expiry(e, i).

Lemma 3.5. *If $\Delta_e^i \ge 0$, then* expiry$(e, i) = t + t'$, *where $t' = \min\{\bar{t} : f(w, \bar{t}) \le \Delta_e^i\}$. Further, given Δ_e^i, the expiry time can be computed in time $O(\log t_{max})$. If $\Delta_e^i = -1$, then* expiry$(e, i) = t$.

Proof. If $\Delta_e^i \ge 0$, meaning s_e^i exists, since $f(w, x)$ is a non-increasing function of x, when x becomes large enough ($\le t_{max}$) we can have $w_{max} \cdot id + f(w, x) - 1 < s_e^i$, i.e., $f(w, x) \le \Delta_e^i$, which further means the range of r_e^{t+x} does not include s_e^i. Since s_e^i is the smallest selected integer in r_e^t at level i, and r_e^{t+x} is the smaller portion of r_e^t and does not include s_e^i, so r_e^{t+x} does not have any selected integer at level i. In other words, e has expired at time $t + x$ as long as $f(w, x) \le \Delta_e^i$. By the definition of the expiry time, we have expiry$(e, i) = t + \min\{x : f(w, x) \le \Delta_e^i\} = t + t'$.

If $\Delta_e^i = -1$, meaning s_e^i does not exist, then there is no integer in r_e^t to be selected at level i. e expires since it was generated at time t, i.e., expiry$(e, i) = t$.

If $\Delta_e^i \ge 0$, we can perform a binary search on the range of $[t, t + t_{max}]$ to find t', using $O(\log t_{max})$ time. If $\Delta_e^i = -1$, simply set expiry$(e, i) = t$. ☐

The pseudocode for ExpiryTime(), which computes expiry(e, i), is formally presented in Algorithm 4.

We can now focus on the efficient computation of Δ_e^i. One possible solution, presented in a preliminary version of this paper [17], is a binary search over the range r_e^t to find Δ_e^i. This approach takes $O(\log w \log t_{max})$ time since in each step of the binary search, a RangeSample [33] operation is invoked, which takes $O(\log w)$ time, and there are $O(\log t_{max})$ such steps in the binary search.

We now present a faster algorithm for computing Δ_e^i, called MinHit() which is described formally in Algorithm 5. Given hash function h and sample level i, in $O(\log w)$ time, MinHit() returns Δ_e^i.

Let Z_p denote the ring of non-negative integers modulo p. Let $l = w_{max} \cdot id$ and $r = w_{max} \cdot id + f(w, 0) - 1$, the left and right end points of r_e^t. The sequence

```
Algorithm 4: ExpiryTime($e, i$)
    Input: $e = (v, w, t, id)$, $i$, $0 \le i \le M$
    Output: $expiry(e, i)$
1   $\Delta_e^i \leftarrow$ MinHit$\left(p, a, h(w_{max} \cdot id), w \cdot f(0) - 1, \lfloor 2^{-i} p \rfloor - 1\right)$ ; /* $h(x) = (ax + b)$
    mod $p$ */
2   if $\Delta_e^i \ge 0$ then                                /* $s_e^i$ exists */
3       $l \leftarrow 0$; $r \leftarrow t_{max}$ ;
4       for $\left(t' \leftarrow \lfloor \frac{l+r}{2} \rfloor; t' \ne l; t' \leftarrow \lfloor \frac{l+r}{2} \rfloor\right)$ do      /* Binary Search for $t'$ */
5           if $(f(w, t') > \Delta_e^i)$ then $l \leftarrow t'$else $r \leftarrow t'$
6       return $t + t'$;
7   else return $t$ ;                                    /* $s_e^i$ does not exist */
```

$h(l), h(l+1), \ldots, h(r)$ is an arithmetic progression over Z_p with a common difference a. The task of finding Δ_e^i reduces to the following problem by setting $u = h(l)$ and $n = f(w, 0) - 1, L = \lfloor p2^{-i} \rfloor - 1$.

Problem 1. *Given integers $p > 0$, $0 \le a < p$, $0 \le u < p$, $n \ge 0$, $L \ge 0$, compute d, which is defined as follows. If set $P = \{j : 0 \le j \le n, (u + j \cdot a) \mod p \le L\} \ne \emptyset$, then $d = \min(P)$; else, $d = -1$.*

Let S denote the following arithmetic progression on Z_p: $\langle u \mod p, (u + a) \mod p, \ldots, (u + n \cdot a) \mod p \rangle$. Let $S[i]$ denote $(u + i \cdot a) \mod p$, the ith number in S. Problem 1 can be restated as: find the smallest integer $j, 0 \le j \le n$, such that $S[j] \le L$.

Note that if $L \ge p$, then obviously $d = 0$. Thus we consider the case $L < p$. Similar to the approach in [33], we divide S into multiple subsequences: $S = S_0 S_1 \ldots S_k$, as follows: $S_0 = \langle S[0], S[1] \ldots, S[i] \rangle$, where i is the smallest natural number such that $S[i] > S[i+1]$. The subsequences S_j, $j > 0$, are defined inductively. If $S_{j-1} = \langle S[t], S[t+1] \ldots, S[m] \rangle$, then $S_j = \langle S[m+1], S[m+2], \ldots, S[r] \rangle$, where r is the smallest number such that $r > m+1$ and $S[r] > S[r+1]$; if no such r exists, then $S[r] = S[n]$. Note that if $S_j = \langle S[t], S[t+1], \ldots, S[m] \rangle$, then $\langle S[t], S[t+1], \ldots, S[m] \rangle$ are in ascending order and if $j > 0$ then $S[t] < a$. Let f_i denote the first element in S_i. Let sequence $F = \langle f_0, f_1, \ldots, f_k \rangle$. Let $|S_i|$ denote the number of elements in S_i, $0 \le i \le k$.

We first observe the critical fact that if $P \ne \emptyset$, $((u + d \cdot a) \mod p)$ must be a member of F. More precisely, we have the following lemma.

Lemma 3.6. *If $d \ne -1$, then $S[d] = f_m \in F$, where $m = \min\{i : 0 \le i \le k, f_i \le L\}$.*

Proof. First, we prove $S[d] \in F$. Suppose $S[d] \notin F$ and $S[d] \in S_t$, for some t, $0 \le t \le k$. Let $f_t = S[d']$. Note that $d' < d$. Since $S[d] \notin F$, we have $f_t \le S[d] \le L$. Because $d' < d$, if d' is not returned, d will not be returned either. This yields a contradiction. Second, we prove $S[d] = f_m$. Suppose $S[d] = f_{m'}$, where $m' > m$. Let

```
Algorithm 5: MinHit($p, a, u, n, L$)
  Input: $p > 0$, $0 \le a < p$, $0 \le u < p$, $n \ge 0$, $L \ge 0$
  Output: $d = \min\{i : 0 \le i \le n, (u + i \cdot a) \bmod p \le L\}$, if $d$ exists; otherwise,
          $-1$.
  // Recursive Call Exit Conditions
1 if ($p \le L$ or $u \le L$) then return 0 ;
2 else if ($a = 0$) then return $-1$ ;
3 else
4     Compute $|S_0|$ ;        /* $S = \{u, (u+a) \bmod p, \cdots, (u + n \cdot a) \bmod p\} = S_0 S_1 \cdots S_k$ */
5     if ($|S_0| = n + 1$) then return $-1$ ;
6     else if ($a = 1$) then return $(p - u)$
  // Recursive Calls
7 $r \leftarrow p \bmod a$ ;
8 Compute $k$, $f_1$ ;        /* $f_1$ is the first element of $S_1$ */
9 if ($a - r \le a/2$) then $d \leftarrow$ MinHit($a, a - r, f_1, k - 1, L$) ;
10 else $d \leftarrow$ MinHit ($a, r, (a - f_1 + L) \bmod a, k - 1, L$) ;
  // Recursive Call Returns
11 if ($d \ne -1$) then
12     Compute $f_{d+1}$ ;
13     $d \leftarrow [(d + 1)p - u + f_{d+1}]/a$ ;
14 return $d$
```

$f_m = S[d']$. Note that $d' < d$ as $m < m'$. Since $d' < d$ and $S[d'] \le L$, if d' is not returned, d will not be returned either. This is also a contradiction. ◻

The next lemma shows that using m and f_m in Lemma 3.6, we can obtain d directly.

Lemma 3.7. *If m exists, then $d = (mp - f_0 + f_m)/a$.*

Proof. Let $S' = S[0], \ldots, S[d]$ denote the sub-sequence starting from f_0 to f_m in S, so $S[0] = f_0$ and $S[d] = f_m$. The distance that has been traveled in the progression over the ring Z_p from f_0 to f_m is $(mp - f_0 + f_m)$. Since the common difference in the progression is a, we have $d = (mp - f_0 + f_m)/a$. Note that $f_0 = u \bmod p$ is known. ◻

The next observation from [33] is crucial for finding m.

Observation 3.2 (Observation 2, Section 3.1 [33]). *Sequence $\bar{F} = F \setminus \{f_0\}$ is an arithmetic progression over Z_a, with common difference $a - r$ (or $-r$ equivalently), where $r = p \bmod a$.*

So, we have two possibilities: (1) If $f_0 \le L$, then $m = 0$ and $f_m = f_0$, thus $d = 0$. (2) Else, the task of finding m is a new instance of Problem 1 of a smaller size by setting:

$$p_{new} = a, a_{new} = a - r, u_{new} = f_1, n_{new} = k - 1, L_{new} = L$$

Note that once m is known, we can directly obtain $f_m = (f_1 + (m-1)(a-r)) \mod a$.

However, because of the similar argument in [33], the reduction may not always be useful since $a - r$ may not be much smaller than a. However, since at least one of $a - r$ or r is less than or equal to $a/2$, we can choose to work with the smaller of $a - r$ or r as follows. The benefit of working with the smaller one will be shown later in the time and space complexity analysis.

Reduction in Case 1: $a - r \leq a/2$. We work with $a - r$. Problem 1 is recursively reduced to a new instance of Problem 1 of a smaller size that finds m over sequence $\bar{F}$ by setting:

$$p_{new} = a, a_{new} = a - r, u_{new} = f_1, n_{new} = k - 1, L_{new} = L$$

Reduction in Case 2: $r < a/2$. We work with r. In this case, things are a bit complex. First we visualize the intuition with the help of Figure 3.2. Note that $\bar{F} = \langle f_1, f_2, \ldots, f_k \rangle$ is a sequence of points lining up along the ring of Z_a with common difference $a - r > a/2$. For simplicity, we only show the first few elements in $\bar{F}$, say $\langle f_1, f_2, \ldots, f_5 \rangle$. We want to find the first point in sequence $\bar{F}$ that is within the dark range $[0, L]$ in Figure 3.2(a).

Note that our goal is to make a_{new} to be r in the parameter setting of the new instance of Problem 1 for finding m, so we flip the ring of Z_a along with the points on it (Figure 3.2(a)) and get the result shown in Figure 3.2(b). After this flipping, the points in $\bar{F}$ comprise a new sequence $\bar{F'} = \langle f'_1, f'_2, \ldots, f'_k \rangle$, where $f'_i = (a - f_i) \mod a$, $1 \leq i \leq k$, the dark range $[0, L]$ is mapped to the new one $[a - L, a - 1] \cup \{0\}$. Note that $\bar{F'}$ is an arithmetic progression over Z_a with common different $-(a - r) \mod a = r$. Let $m' = \min\{i : a - L \leq f'_i \leq a - 1 \text{ or } f'_i = 0, 1 \leq i \leq k\}$, i.e., f'_m is the first point in $\bar{F'}$ such that f'_m is within the dark range in Figure 3.2(b). Obviously $m' = m$, as we did not change the relative positions of all the points and the dark range during the flipping. Note that the idea of flipping the ring is implicitly proposed in [33], however, it is not clear how to further apply the technique in [33] to find m'.

Our new idea is to shift the origin of the ring of Z_a in Figure 3.2(b) by a distance of L in a counter-clockwise direction without moving all the points and the dark range, resulting in Figure 3.2(c). After this shifting, sequence $\bar{F'}$ in Figure 3.2(b) is mapped to a new sequence $\bar{F''} = \langle f''_1, f''_2, \ldots, f''_k \rangle$ in Figure 3.2(c), where $f''_i = (f'_i + L) \mod a$, and the dark range in Figure 3.2(b) is mapped to $[0, L]$ in Figure 3.2(c). Let $m'' = \min\{i : 0 \leq f''_i \leq L, 1 \leq i \leq k\}$, i.e., $f_{m''}$ is the first point in $\bar{F''}$ such that $f_{m''}$ is within the dark range $[0, L]$ in Figure 3.2(c). Obviously $m'' = m'$, as we did not change the relative positions of all the points and the dark range during the shifting of the origin in Figure 3.2(b). This further implies $m'' = m$. Therefore, Problem 1 can be recursively reduced to a smaller problem of finding m'' over sequence $\bar{F''}$ by setting:

$$p_{new} = a, a_{new} = r, u_{new} = (a - f_1 + L) \mod a, n_{new} = k - 1, L_{new} = L$$

We note that the idea of shifting the origin of the ring is quite simple and useful. Using this idea simplifies the *Hits* algorithm in [33] since all the additional operations dealing with the effect of flipping the ring can be omitted.

The above visualized intuition in case 2 is validated by the following lemma.

LEMMA 3.8. *Given p, a, u, n, L as in Problem 1, set $P = \{i : 0 \le i \le n, (u + i \cdot a) \mod p \le L\}$ and $P' = \{j : 0 \le j \le n, ((p - u + L) \mod p + j \cdot (p - a)) \mod p \le L\}$, then:*

$$P = P'$$

Proof. (i) $P \subseteq P'$. Suppose $\gamma \in P$, then $0 \le \gamma \le n$ and $(u + \gamma \cdot a) \mod p \le L$. We prove $\gamma \in P'$.

$$\begin{aligned} &[(p - u + L) \mod p + \gamma \cdot (p - a)] \mod p \\ &= [p - u + L + \gamma \cdot (p - a)] \mod p \\ &= [L - (u + \gamma \cdot a)] \mod p \\ &= [L - (u + \gamma \cdot a) \mod p] \mod p \end{aligned}$$

Since $0 \le (u + \gamma \cdot a) \mod p \le L$, we have $0 \le [L - (u + \gamma \cdot a) \mod p] \mod p \le L$. Thus $\gamma \in P'$.

(ii) $P' \subseteq P$. Suppose $\gamma \in P'$, then $0 \le \gamma \le n$ and $[(p-u+L) \mod p + \gamma \cdot (p-a)] \mod p \le L$. We prove $\gamma \in P$.

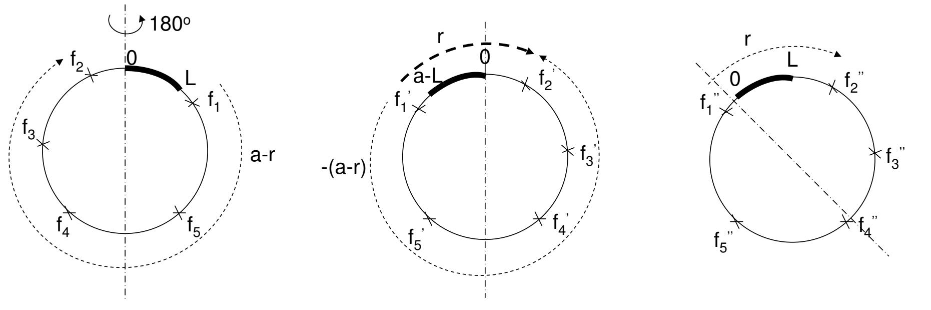
$$\begin{aligned} &[(p - u + L) \mod p + \gamma \cdot (p - a)] \mod p \\ &= [L - (u + \gamma \cdot a) \mod p] \mod p \\ &\le L \end{aligned}$$

If $(u + \gamma \cdot a) \mod p > L$, say $(u + \gamma \cdot a) \mod p = L + \sigma < P$ for some $\sigma > 0$, from the above inequality, we can have that $(-\sigma) \mod p = p - \sigma \le L$, i.e., $L + \sigma \ge P$, this yields a contradiction. Therefore, $(u + \gamma \cdot a) \mod p \le L$. So, $\gamma \in P$. □

Since $P = P'$, then the Problem 1 with the setting $p_{new} = a$, $a_{new} = a - r$, $u_{new} = f_1$, $n_{new} = k - 1$, $L_{new} = L$ and the Problem 1 with the setting $p_{new} = a$, $a_{new} = r$, $u_{new} = (a - f_1 + L) \mod a$, $n_{new} = k - 1, L_{new} = L$ return the same answer.

LEMMA 3.9. *The algorithm* MinHit(p, a, u, n, L) *(shown in Algorithm 5) computes d in Problem 1 in time $O(\log n)$ and space $O(\log p + \log n)$.*

Proof. **Correctness.** Recall that MinHit(p, a, u, n, L) should return $d = \min\{i : 0 \le i \le n, (u + i \cdot a) \mod p \le L\}$, if such d exists; otherwise, return $d = -1$. Clearly,



(a) Ring of Z_a and Sequence $\bar{F}$ (b) Flipped Ring of Z_a and Sequence $\bar{F}'$ (c) Flipped Ring of Z_a with Shifted Origin and Sequence $\bar{F}''$

FIG. 3.2. *Find $f_m \in \bar{F}$ over the Ring of Z_a in the Case of $r < a/2$*

if $p \le L$ or $u \le L$, $d = 0$. Line 1 correctly handles this scenario; Else, if $a = 0$, which means all the integers in sequence S are equal to u, since after line 1 we know $u > L$, $d = -1$ is returned in line 2; Else, if $S = S_0$, since after line 1 we know $f_0 > L$, all the integers in S are greater than L, thus $d = -1$ is returned at line 5; Else, if $a = 1$, since $|S| > |S_0|$, we can easily find $f_1 = S[p - u] = 0 \le L$, thus $d = p - u$ is returned by line 6. If all the above conditions are not satisfied, we have $a > 1, u > L, L < p, |S| > |S_0|$. Since $f_0 = u > L$, by Lemma 3.6, if $d \ne -1$, we know $S[d] \in \bar{F}$. Because of Observation 3.2, we can make a recursive call at lines 9 or 10, to find j, $1 \le j \le k$, such that $f_j = S[d]$. Because of Lemma 3.8, lines 9 and 10 return the same result (with different time cost though). Using the formula presented in Lemma 3.7, the answer for the original problem is calculated and returned by lines 11–14 using the answer from the recursive call at either step 9 or 10. Therefore, MinHit(p, a, u, n, L) correctly returns d as the answer for Problem 1.

Time Complexity. We assume that the additions, multiplications and divisions take unit time. It is clear that lines 1–8 and 11–14 can be computed in constant time. In each recursive call at lines 9 and line 10, because $n_{new} \le \lceil n \cdot a/p \rceil$ and $a \le p/2$ always hold in every recursive call, thus we have $n_{new} \le n/2$, which yields the time cost of MinHit(p, a, u, n, L) is $O(\log n)$.

Space Complexity. In each recursive call, MinHit() needs to store a constant number of local variables such as $p, a, n, etc.$. Since p dominates a, u and L (if $L \ge p$, then MinHit() returns without recursive calls), each recursive call needs $O(\log p + \log n)$ stack space. Since the depth of the recursion is no more than $\log n$, the space cost is $O(\log n(\log p + \log n))$. Using a similar argument as in [33], in general MinHit$(p_1, p_2, p_3, p_4, p_5) = \beta + \gamma\,$MinHit$(p'_1, p'_2, p'_3, p'_4, p'_5)$, where β and γ are functions of $p_1, \ldots, p_5$. This procedure can be implemented using tail recursion, which does not need to allocate space for the stack storing the state of each recursive step and does not need to tear down the stack when it returns. Thus, the space cost can be reduced

Algorithm 6: DecayedSumQuery(c)

```
1 $\ell = \min\{i : 0 \leq i \leq M, t_i \leq c\}$ ;
2 if $\ell$ does not exist then return ; // the algorithm fails
3 if $\ell$ exists then return $\frac{1}{p_\ell}\sum_{e \in S_\ell} RangeSample(r_e^c, \ell)$;
```

to $O(\log p + \log n)$. ☐

Theorem 3.10. *Given a stream element $e = (v, w, t, id)$ and the sample level i, $0 \leq i \leq M$,* expiry(e, i) *can be computed in time $O(\log w + \log t_{max})$ using space $O(\log p + \log w)$.*

Proof. MinHit() can compute Δ_e^i in $O(\log w)$ time using space $O(\log p + \log w)$. Due to Lemma 3.5, given Δ_e^i algorithm ExpiryTime() computes expiry(e, i) using additional $O(\log t_{max})$ time for the binary search. ☐

Faster Computation of Expiry Time. In some cases, the expiry time can be computed faster than using the above algorithm. In particular, it can be computed in $O(\log w)$ time, if the decay function f has the following property: given an initial weight w and decayed weight $w' \leq w$, $\min\{x : f(w, x) = w'\}$ can be computed in a constant number of steps. This includes a large class of decay functions. For example, for the integral version of exponential decay $f(w, x) = \lfloor \frac{w}{a^x} \rfloor$, given $\Delta_e^i \geq 0$ (note that $w' = \Delta_e^i + 1$), which is computed $O(\log w)$ time, the expiry time can be computed in a constant number of steps through expiry$(e, i) = \left\lfloor \log_a \frac{w}{\Delta_e^i + 1} \right\rfloor + t + 1$, where $e = (v, w, id, t)$. A similar observation is true for the integral version of polynomial decay $f(w, x) = \lfloor w \cdot (x + 1)^{-a} \rfloor$. For the sliding window decay, given $\Delta_e^i \geq 0$, then expiry$(e, i) = t + W$, where $e = (v, w, t, id)$ and W is the window size.

3.4. Computing Decayed Aggregates Using the Sketch. We now describe how to compute a variety of decayed aggregates using the sketch S. For $i = 0 \ldots M$, 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:

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

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 3.11. *For any integral decay function f, Algorithm 6 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 $O(\log w(\log \frac{1}{\epsilon} + \log w + \log t_{max}))$. The space complexity is $O(\frac{1}{\epsilon^2}(\log w_{max} + \log id_{max}))$.*

Proof. We show the correctness of our algorithm for the sum through a reduction to the range-efficient algorithm for counting distinct elements from [33] (we refer to this algorithm as the PT algorithm, for the initials of the authors of [33]). 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 $|\{e \in D : \text{RangeSample}(r_e^c, \ell') > 0\}| \leq \tau$, and returns an estimate $Y = \frac{1}{p_{\ell'}} \sum_{e\in D} \text{RangeSample}(r_e^c, \ell')$. From Theorem 1 in [33], 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 Algorithm 6 to answer a query for the sum. Suppose ℓ exists, then ℓ is the smallest integer such that $t_\ell \leq c$. 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 $|\{e : \text{RangeSample}(r_e^c, \ell) > 0\}| \leq \tau$. 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 $i, 0 \leq i \leq M$, $|\{e \in D : \text{RangeSample}(r_e^c, i) > 0\}| > \tau$, 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 Theorem 3.4. $\square$

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. Now we consider the estimation of the selectivity

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

where $P(v,w)$ is a predicate given at the query time. We return the selectivity of sample S_ℓ using the predicate P as the estimate of Q, where S_ℓ is the lowest numbered sample that does not have any discarded element whose expiry time is larger than c. The formal algorithm is given in Algorithm 7. We show that by setting $\tau = 492/\epsilon^2$ and $M = \lceil \log w_{max} + \log id_{max} \rceil$, we can get Theorem 3.26.

The following process only helps visualize the proof, and is not executed by the algorithm. Since the sketch is duplicate insensitive (Lemma 3.2), we simply consider stream D, which is the set of distinct elements in stream R. At query time c, stream D is converted to be a stream of intervals $D' = \{r_d^c : d \in D\}$. Note that $d = (v, w, t, id)$ and $r_d^c = [w_{max} * id, w_{max} * id + f(w, c-t) - 1]$. Further, stream D' is expanded to stream I of the constituent integers. For each interval $[x, y] \in D'$, stream I consists of $x, x+1, \ldots, y$. Clearly all the items in I are distinct and the decayed sum $V = |I|$. Given the selectivity predicate $P(v, w)$, let $\hat{I} = \{x \in r_d^c) : d = (v, w, t, id) \in D, p(v, w) = 1\}$ and $V' = |I'|$. Note that $I' \subseteq I$ and the selectivity with predicate $P(v, w)$ is $Q = \frac{V'}{V}$, for which we compute an estimate Q'. Recall that the sample size $\tau = C/\epsilon^2$, where C is a constant to be determined through the analysis.

The next part of this section, from Fact 3.1 through Lemma 3.25, helps in the proof of Theorem 3.26 (stated formally below).

FACT 3.1 (Fact 1 in [33]). *For any $i \in [0 \ldots M]$, $\frac{1}{2^{i+1}} \leq p_i \leq \frac{1}{2^i}$*

LEMMA 3.12. *If $|D'| \leq \tau$, then $Q = Q'$*

Proof. If $|D'| \leq \tau$, all the $r_d^c \in D'$ can be implicitly stored in S_0, i.e., all unexpired stream elements can be stored in S_0, which can return the exact Q. □

Thus, in the following part of the proof, we assume $|D'| > \tau$.

DEFINITION 3.13. *For each $e \in I$, for each level $i = 0, 1, \ldots, M$, random variable $x_i(e)$ is defined as follows: if $h(e) \in [0, \lfloor p2^{-i} \rfloor]$, then $x_i(e) = 1$; else $x_i(e) = 0$.*

DEFINITION 3.14. *For $i = 0, 1, \ldots, M$, T_i is the set constructed by the following probabilistic process. Start with $T_i \leftarrow \emptyset$. If there exists at least one integer $y \in r_d^c$, where $d \in D$, such that $x_i(y) = 1$, insert d into T_i.*

Note that T_i is defined for the purpose of the proof only, but the T_is are not stored by the algorithm. For each level i, the algorithm only stores at most τ elements with largest expiry time.

DEFINITION 3.15. *For $i = 0, 1, \ldots, M$, $X_i = \sum_{y \in r_d^c} x_i(y)$, $X_i' = \sum_{y \in r_d^c, p(v,w)=1} x_i(y)$, where $d = (v, w, t, id) \in D$.*

LEMMA 3.16. *For any $e \in r_d^c, d \in D$, $\mathsf{E}[x_i(e)] = p_i$, $\sigma^2_{x_i(e)} = p_i(1-p_i)$, $0 \leq i \leq M$.*

Proof. $\mathsf{E}[x_i(e)] = \Pr[x_i(e) = 1] = \Pr[0 \leq h(e) \leq \lfloor p2^{-i} \rfloor] = \lfloor p2^{-i} \rfloor = p_i$.

$\sigma^2_{x_i(e)} = \mathsf{E}[x_i^2(e)] - \mathsf{E}[x_i(e)]^2 = \Pr[x_i^2(e) = 1] - \Pr[x_i(e) = 1]^2 = p_i - p_i^2 = p_i(1-p_i)$

□

LEMMA 3.17. *For $i = 0, 1, \ldots, M$, $\mathsf{E}[X_i] = p_i V$, $\sigma^2_{X_i} = p_i(1-p_i)V$, $\mathsf{E}[X_i'] = p_i V'$, $\sigma^2_{X_i'} = p_i(1-p_i)V'$*

Proof. $\mathsf{E}[X_i] = \mathsf{E}[\sum_{y \in r_d^c} x_i(y)] = |\{y \in r_d^c : d \in D\}| \cdot \mathsf{E}[x_i(y)] = p_i V$. Since $x_i(y)$'s are pairwise independent random variables, we have: $\sigma^2_{X_i} = |\{y \in r_d^c : d \in D\}| \cdot \sigma^2_{x_i(y)} = p_i(1-p_i)V$. Similarly, $\mathsf{E}[X_i'] = p_i V'$, $\sigma^2_{X_i'} = p_i(1-p_i)V'$ are true. □

DEFINITION 3.18. *For $i = 0, 1, \ldots, M$, define event B_i to be true if $Q' \notin [Q - \epsilon, Q + \epsilon]$, and false otherwise; define event G_i to be true if $(1 - \epsilon/2)p_i V \leq X_i \leq (1 + \epsilon/2)p_i V$, false otherwise.*

DEFINITION 3.19. *Let $\ell^* \geq 0$ be an integer such that $\mathsf{E}[X_{\ell^*}] \leq \tau/2$ and $\mathsf{E}[X_{\ell^*}] > \tau/4$.*

LEMMA 3.20. *Level ℓ^* is uniquely defined and exists for every input stream D.*

Proof. Since $|D'| > \tau$, $\mathsf{E}[X_0] > \tau$. By the definition of $M = \log w_{max} id_{max}$, it must be true that $V < 2^M$ for any input stream D, so that $\mathsf{E}[X_M] \leq 1$. Since for every increment in i, $\mathsf{E}[X_i]$ decreases by a factor of 2, there must be a unique level $0 < \ell^* < M$ such that $\mathsf{E}[X_{\ell^*}] \leq \alpha/2$ and $\mathsf{E}[X_{\ell^*}] \geq \alpha/4$. □

From now on we consider the case with $0 < Q \leq 1/2$. By symmetry, a similar proof exists for the case with $1/2 \leq Q < 1$. Obviously the algorithm can return $Q' = Q$, if $Q \in \{0, 1\}$.

The following lemma shows that for levels that are less than or equal to ℓ^*, Q' is very likely to be close to Q.

LEMMA 3.21. *For $0 \leq \ell \leq \ell^*$,*

$$\Pr[B_\ell] < \frac{5}{C \cdot 2^{\ell^* - \ell - 4}}$$

Proof.

$$\begin{aligned} \Pr[B_\ell] &= \Pr[G_\ell \wedge B_\ell] + \Pr[\bar{G}_\ell \wedge B_\ell] \\ &\leq \Pr[B_\ell | G_\ell] \cdot \Pr[G_\ell] + \Pr[\bar{G}_\ell] \leq \Pr[B_\ell | G_\ell] + \Pr[\bar{G}_\ell] \end{aligned} \tag{3.1}$$

Using Lemmas 3.22 and 3.23 in Equation 3.1, we get:

$$\Pr[B_\ell] < \frac{5}{C \cdot 2^{\ell^* - \ell - 4}}$$

□

LEMMA 3.22. *For $0 \leq \ell \leq \ell^*$,*

$$\Pr[\bar{G}_\ell] < \frac{1}{C \cdot 2^{\ell^* - \ell - 4}}$$

Proof. By Lemma 3.17, $\mu_{X_\ell} = p_\ell V, \sigma^2_{X_\ell} = p_\ell(1 - p_\ell)V$, and by Chebyshev's inequality, we have

$$\begin{aligned} \Pr[\bar{G}_\ell] &= \Pr[X_\ell < (1 - (\epsilon/2))\mu_{X_\ell} \vee X_\ell > (1 + (\epsilon/2))\mu_{X_\ell}] \\ &= \Pr[|X_\ell - \mu_{X_\ell}| > (\epsilon/2) \cdot \mu_{X_\ell}] \\ &\leq \frac{\sigma^2_{X_\ell}}{(\epsilon/2)^2 \mu^2_{X_\ell}} = (1 - p_\ell) / \left((\epsilon/2)^2 \cdot \mu_{X_\ell}\right) \\ &\leq \frac{1}{(\epsilon/2)^2 \cdot \mu_{X_\ell}} \leq \frac{1}{C \cdot 2^{\ell^* - \ell - 4}} \end{aligned}$$

The last inequality is due to the fact: $\mu_{X_\ell} \geq 2^{\ell^* - \ell} \cdot \mu_{X^*_\ell} > 2^{\ell^* - \ell} \cdot \tau/4 = 2^{\ell^* - \ell - 2} \cdot C/\epsilon^2$, using Fact 3.1 □

LEMMA 3.23. *For $0 \leq \ell \leq \ell^*$,*

$$\Pr[B_\ell | G_\ell] < \frac{1}{C \cdot 2^{\ell^* - \ell - 6}}$$

Proof.

$$\Pr[B_\ell|G_\ell] = \Pr[Q < Q' - \epsilon|G_\ell] + \Pr[Q > Q' + \epsilon|G_\ell]$$

The proof will consist of two parts, Equations 3.2 and 3.3.

$$\Pr[Q + \epsilon < Q'|G_\ell] < \frac{1}{C \cdot 2^{\ell^* - \ell - 5}} \tag{3.2}$$

$$\Pr[Q - \epsilon > Q'|G_\ell] < \frac{1}{C \cdot 2^{\ell^* - \ell - 5}} \tag{3.3}$$

Proof of Equation 3.2: Let $Y = \sum_{y \in I'} x_\ell(y) = Q'X_\ell > (Q + \epsilon)X_\ell$. By Lemma 3.17, we have $\mu_Y = p_\ell VQ$, $\sigma_Y^2 = p_\ell(1 - p_\ell)VQ$. Using Chebyshev's inequality and the fact $X_\ell \geq (1 - \epsilon/2)p_\ell V$, we have the following,

$$\begin{aligned}
\Pr[Q + \epsilon < Q'|G_\ell] &\leq \Pr[Y > (Q + \epsilon)X_\ell|G_\ell] \\
&= \Pr[(Y > (Q + \epsilon)X_\ell) \wedge G_\ell] / \Pr[G_\ell] \\
&\leq \Pr[Y > (Q + \epsilon)(1 - \epsilon/2)p_\ell V] / \Pr[G_\ell] \\
&= \Pr[Y - \mu_Y > (Q + \epsilon)(1 - \epsilon/2)p_\ell V - \mu_Y] / \Pr[G_\ell] \\
&\leq \left(\frac{\sigma_Y^2}{[(Q + \epsilon)(1 - \epsilon/2)p_\ell V - \mu_Y]^2} \right) / \Pr[G_\ell] \\
&= \left(\frac{p_\ell(1 - p_\ell)VQ}{[(Q + \epsilon)(1 - \epsilon/2)p_\ell V - p_\ell VQ]^2} \right) / \Pr[G_\ell] \\
&\leq \left(\frac{4}{\epsilon^2 p_\ell V} \right) / \Pr[G_\ell] < \left(\frac{1}{C \cdot 2^{\ell^* - \ell - 4}} \right) / \left(1 - \frac{1}{C \cdot 2^{\ell^* - \ell - 4}} \right) \\
&< \frac{1}{C \cdot 2^{\ell^* - \ell - 5}}
\end{aligned}$$

The last three inequalities use the facts: $(1 - p_\ell)Q < 1$, $(Q + \epsilon)(1 - \epsilon/2) \geq Q + \epsilon/2$ due to $0 < \epsilon < Q \leq 1/2$, $p_\ell V > 2^{\ell^* - \ell}\tau/4$ and choosing $C \geq 32$.

Proof of Equation 3.3: By symmetry, the proof is similar as the one for Equation 3.2. Therefore,

$$\Pr[B_\ell|G_\ell] < \frac{1}{C \cdot 2^{\ell^* - \ell - 6}}$$

□

Lemma 3.24.

$$\sum_{\ell=0}^{\ell=\ell^*} \Pr[B_\ell] < \frac{160}{C}$$

Proof. The proof directly follows from Lemma 3.21.

$$\sum_{\ell=0}^{\ell=\ell^*} \Pr[B_\ell] = \sum_{\ell=0}^{\ell=\ell^*} \frac{5}{C \cdot 2^{\ell^* - \ell - 4}} = \frac{80}{C} \sum_{i=0}^{\ell^*} \frac{1}{2^i} < \frac{160}{C}$$

□

Lemma 3.25.

$$\Pr[\ell > \ell^*] < \frac{4}{C}$$

Proof. If $\ell > \ell^\star$, it follows that $X_{\ell^\star} > |T_{\ell^\star}| > \tau$, else the algorithm would have stopped at a level less than or equal to $\ell^\star$. Thus, $\Pr[\ell > \ell^\star] \le \Pr[X_{\ell^\star} > \tau]$. Let $Y = X_{\ell^\star}$. By Lemma 3.17, Chebyshev's inequality and the fact $\mu_Y < \tau/2$, we have

$$\Pr[\ell > \ell^\star] \le \Pr[Y > \tau] \le \Pr[Y > 2\mu_Y] = \Pr[Y - \mu_Y > \mu_Y] = \frac{\sigma_Y^2}{\mu_Y^2} = \frac{p_\ell(1-p_\ell)V}{p_\ell^2 V^2} = \frac{1-p_\ell}{p_\ell V}$$

Since $\mu_Y = p_\ell V > \tau/4$, we have

$$\Pr[\ell > \ell^\star] \le \frac{1-p_\ell}{\tau/4} < 4/\tau = \frac{4}{C}\epsilon^2 < \frac{4}{C}$$

□

Theorem 3.26. *For any integral decay function f, Algorithm 7 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 $O(\log w(\log \frac{1}{\epsilon} + \log w + \log t_{max}))$. The space complexity is $O(\frac{1}{\epsilon^2}(\log w_{max} + \log id_{max}))$.*

Proof. Let f denote the probability that the algorithm fails to return an ϵ-approximate selectivity estimation of D. Using Lemmas 3.24 and 3.25, we get:

$$\begin{aligned} f &= \Pr[\ell > M] + \Pr[\bigcup_{i=0}^{M}(\ell = i) \wedge B_i] \\ &\le \Pr[\ell > \ell^\star] + \sum_{i=0}^{\ell^\star}\Pr[B_i] < \frac{164}{C} = \frac{1}{3}, \text{ by choosing } C = 492 \end{aligned}$$

The query time complexity analysis is similar to the one for the sum in Theorem 3.11. The expected time for each update and the space complexity directly follows from Theorem 3.4. □

As in the sum case, 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 3.27. *For any integral decay function f, it is possible to answer queries for ϵ-approximate ϕ-quantiles and frequent items queries using the sketch, in time $O(\log M + \frac{1}{\epsilon^2}\log(\frac{w_{max}}{\epsilon}))$. The expected time for each update is $O(\log w(\log\frac{1}{\epsilon} + \log w + \log t_{max}))$. The space complexity is*
$O(\frac{1}{\epsilon^2}(\log w_{max} + \log id_{max}))$.

Proof. The expected time for each update and the space complexity directly follows from Theorem 3.4. Now we show how to reduce a sequence of problems to

Algorithm 7: DecayedSelectivityQuery(P,c)

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

instances of selectivity estimation. To answer the query for the aggregate of interest, we first find the appropriate weighted sample S_ℓ in $\log M$ time, where ℓ is defined (as before) as the smallest integer such that $t_\ell < c$.

- **Rank.** 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 $P_{\le\nu}$ 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.
- **Median.** The median is the item whose rank is 0.5. To find the median, we can sort S_ℓ by value in $O(\tau \log \tau)$ time, then evaluate the rank of every distinct value in the sample and return the median of S_ℓ as the median of the stream with an additional $O(\tau \log w_{max})$ time cost. Due to the argument about the rank estimation, we have that the median of S_ℓ has a rank of 0.5 over the stream with additive error at most ϵ with probability at least $1-\delta$.
- **Quantiles.** Quantiles generalize the median to find items whose ranks are multiples of ϕ, e.g. the quintiles, which are elements at ranks 0.2, 0.4, 0.6 and 0.8. Again, sort S_ℓ by value and return the ϕ-quantile of S_ℓ as the ϕ-quantile of the stream with additive error at most ϵ with probability at least $1-\delta$. The argument is similar to the one for the median.
- **Frequent items.** Sort S_ℓ in $O(\tau \log \tau)$ time, then evaluate the frequency of every distinct value in S_ℓ with another $O(\tau \log w_{max})$ time cost. We can return those values whose frequency in S_ℓ is ϕ or more as the frequent items in the stream, because for each returned value ν, regarding the predicate "$P_{=\nu}(v,w) = 1$ if $v = \nu$", the selectivity of ν, which is also the frequency of ν, in the stream is ϕ or more with additive error at most ϵ with probability at least $1-\delta$.

□

4. Decomposable 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 $f_W(w,x) = w$ if $x < W$, and $f_W(w,x) = 0$ 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. We can compute the expiry time of any element e at level ℓ in $\log w$ time as $(t+W)$ if $\Delta_e^\ell \ge 0$; t, otherwise. We can prove a stronger result though: If we set $f(w,x) = w$ for all $x \ge 0$ when inserting the element (i.e., element e never expires at level ℓ) unless

$\Delta_e^\ell < 0$, 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 sliding window decayed sum, V, such that $\Pr[|\hat{V} - V| < \epsilon V] > \frac{2}{3}$ and an estimate $\hat{Q}$ of the sliding window decayed selectivity, Q, such that $\Pr[|\hat{Q} - Q| < \epsilon] > \frac{2}{3}$. The time to answer either query is $O(\log M + \tau)$.*

Proof. Observe that for all parameters W, at any level ℓ, over the set of element $e = (v, w, t, id)$ where $\Delta_e^\ell \geq 0$, 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 e with $\Delta_e^\ell \geq 0$ arrives in level i. If there are fewer than τ items at the level, we retain it. Otherwise, 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 3.11 and 3.26.

At the time of the query, for the selected sample, we need to compute the contribution of each range to the aggregate – this can be done through a call to the RangeSample routine. We can make the query time smaller at the cost of increased processing time per element (but the same asymptotic complexity for the processing time per element) by calling the RangeSample routine during insertion, and not needing to recompute this at the query time. This yields the desired time complexity of processing an element and of the query time. □

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 a Decomposable Decay Function to Sliding Window Decay. In this section, we show that for any decomposable decay function of the form $f(w, x) = w \cdot g(x)$, the computation of decayed aggregates can be reduced to the computation of aggregates over sliding window decay. This randomized reduction generalizes a (deterministic) idea from Cohen and Strauss [14]: 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.

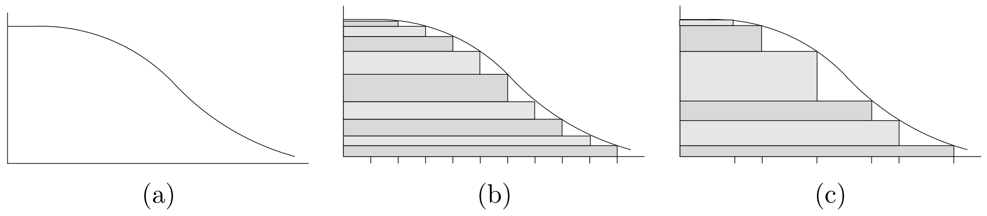


FIG. 4.1. *Reduction of a decomposable decay function to sliding window: (a) a sample decay function (b) breaking the decay function into sliding windows every time step (c) computing sliding windows only for the subset of stored timestamps.*

Selectivity Estimation. LEMMA 4.2. *Selectivity estimation using any decomposable decay function $f(w,x) = w \cdot g(x)$ can be rewritten as the combination of at most $2c$ sliding window queries, where c is the current time.*

Proof. 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

$$Q = \sum_{(v,w,t,id)\in D} w \cdot P(v,w) \cdot g(c-t) / \sum_{(v,w,t,id)\in D} w \cdot g(c-t), \tag{4.1}$$

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

$$A = g(c-t_1)\sum_{i=1}^{n} w_i P(v_i, w_i) + \sum_{t=t_1+1}^{t_n}\left([g(c-t) - g(c-t+1)] \cdot \sum_{\{i: t_i \geq t\}} P(v_i, w_i) w_i\right)$$

$$B = g(c-t_1)\sum_{i=1}^{n} w_i + \sum_{t=t_1+1}^{t_n}\left([g(c-t) - g(c-t+1)] \cdot \sum_{\{i: t_i \geq t\}} w_i\right)$$

We compute A and B separately; first, consider B, which is equivalent to V, the decayed sum under the function $w \cdot g(x)$. Write V^W for the decayed sum under the sliding window of size W. We can compute $\hat{V} = \sum_{t=t_1+1}^{t_n} ([g(c-t) - g(c-t+1)] \cdot V^{c-t})$, 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 $(\sum_i w_i) g(c-t_1)$, 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 union bound. Similarly, A can also be computed by using the sliding window algorithm for the sum. Further, the data stream over which A is computed is a substream, which satisfies the selectivity predicate, of D, over which B is computed. Thus theorem 4.1 implies each sliding window query in A is accurate up to a $(\pm\epsilon V)$ additive error with probability at least $1 - \delta$. This analysis 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 $O(\log c)$. □

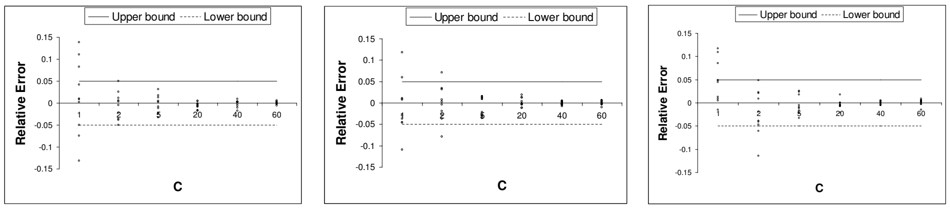
THEOREM 4.3. *We can answer a selectivity query using an arbitrary decomposable decay function $f(w, x) = w \cdot g(x)$ in time $O(M\tau \log(\frac{M\tau}{\delta}) \log(M\tau \log \frac{M\tau}{\delta}))$ to find $\hat{Q}$ so that* $\Pr[|Q - \hat{Q}| > \epsilon] < \delta$.

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 (4.1) by only probing at these timestamps, as:

$$\sum_{i=0}^{M} \sum_{\substack{j=1 \\ t_i^j < t_{i-1}^{min}}}^{|S_i|} (g(c - t_i^j) - g(c - t_i^{j+1}))V^{c - t_i^j} \tag{4.2}$$

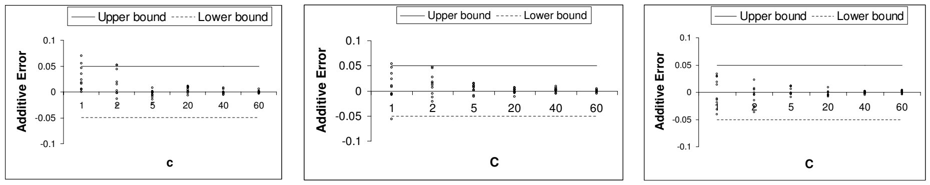
where for $0 \leq i \leq M$, t_i^{min} denotes the smallest (oldest) timestamp of the items in S_i, and $t_{-1}^{min} = c + 1$, where c is the current time (this avoids some double counting issues). This process is illustrated in Figure 4.1: we show the original decay function, and estimation at all 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 $g(c - t_i^j) - g(c - t_i^{j+1})$.

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. □



(a) Exponential Decay, $\beta = 0.01$ (b) Polynomial Decay, $\alpha = 1.0$ (c) Sliding Window, $W = 200$s

FIG. 4.2. *Decayed Sum: Accuracy vs C ($\epsilon = 0.05$)*



(a) P_1: $P_1(v, w) = 1$ iff $v/w \geq 2$ (b) P_2: $P_2(v, w) = 1$ iff $v/w \geq 3$ (c) P_3: $P_3(v, w) = 1$ iff $v/w \geq 4$

FIG. 4.3. *Selectivity with Exponential Decay: Accuracy vs C ($\epsilon = 0.05, \beta = 0.01$)*

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 4.4. *The estimation of decayed sum using an arbitrary decomposable decay function can be rewritten as the combination of at most c sliding window queries, where c is the current time.*

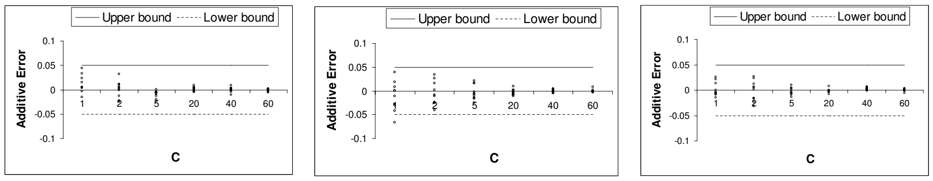
THEOREM 4.5. *We can answer a query for the sum using an arbitrary decomposable decay function $f(w, x) = w \cdot g(x)$ in time $O(M\tau \log(\frac{M\tau}{\delta}) \log(M\tau \log(\frac{M\tau}{\delta})))$ to find $\hat{V}$ such that $\Pr[|\hat{V} - V| > \epsilon V] < \delta]$.*

5. Experiments. In this section, we experimentally evaluate the space and time costs of the sketch, as well as its accuracy in answering queries. We consider three popular integral decay functions: sliding window decay, and modified versions of polynomial and exponential decay. The decay functions are defined as follows:

(1) Sliding window decay with window size W: $f_W(w, x) = w$ if $x \leq W$, and 0 otherwise. We experiment over a range of window sizes, ranging from 200 seconds to 25 hours.

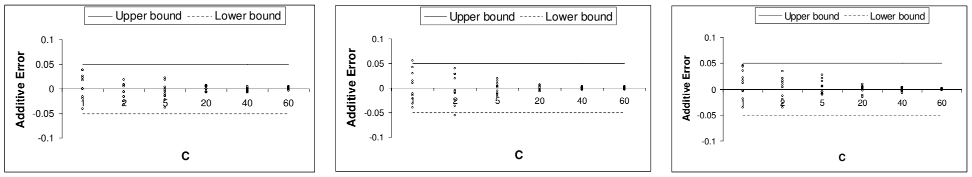
(2) Polynomial decay: $f(w, x) = \left\lfloor \frac{w}{(x+1)^\alpha} \right\rfloor$. We use $\alpha \in \{1.0, 1.5, 2.0, 2.5, 3\}$

(3) Exponential decay: $f(w, x) = \left\lfloor \frac{w}{e^{\beta x}} \right\rfloor$. We use $\beta \in \{0.01, 0.2, 0.4, 0.6, 0.8\}$



(a) P_1: $P_1(v, w) = 1$ iff $v/w \geq 2$ (b) P_2: $P_2(v, w) = 1$ iff $v/w \geq 3$ (c) P_3: $P_3(v, w) = 1$ iff $v/w \geq 4$

FIG. 4.4. *Selectivity with Polynomial Decay: Accuracy vs C ($\epsilon = 0.05, \alpha = 1.0$)*



(a) P_1: $P_1(v, w) = 1$ iff $v/w \geq 2$ (b) P_2: $P_2(v, w) = 1$ iff $v/w \geq 3$ (c) P_3: $P_3(v, w) = 1$ iff $v/w \geq 4$

FIG. 4.5. *Selectivity with Sliding Window: Accuracy vs C ($\epsilon = 0.05$, $W = 200$ seconds)*

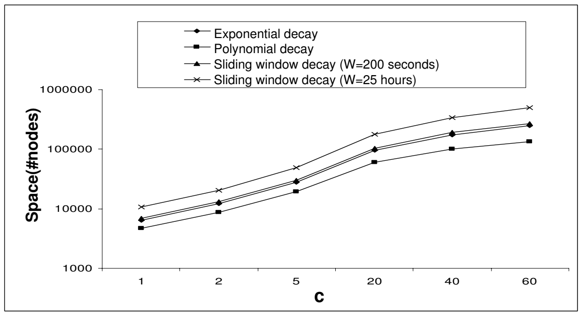
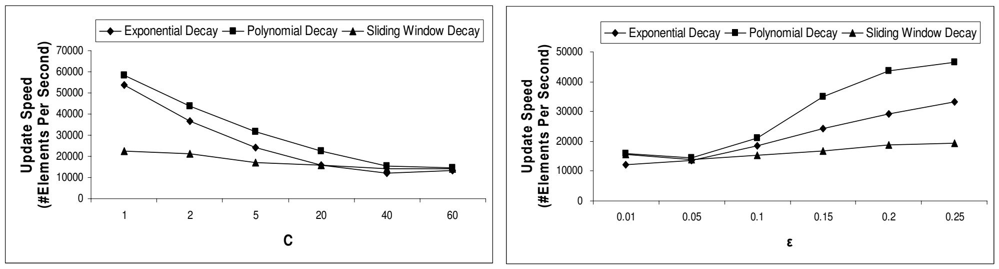
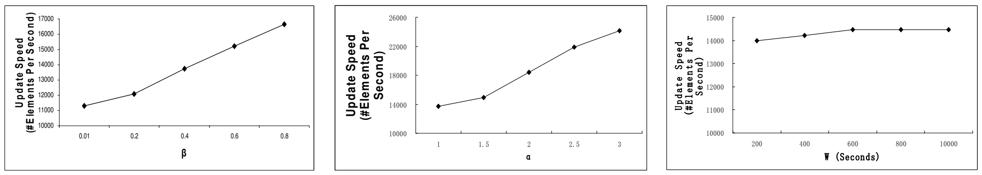


FIG. 4.6. *Space vs C ($\epsilon = 0.05$, $\alpha = 1.0$, $\beta = 0.01$, $W = 200$ seconds, 25 hours)*



(a) Update Speed vs C ($\epsilon = 0.05$, $\alpha = 1.0$, $\beta = 0.01$, $W = 200$ seconds) (b) Update Speed vs ϵ ($C = 60, \beta = 0.01, \alpha = 1.0, W = 200$ Seconds)

FIG. 4.7. *Update speed for different decay functions*



(a) Exponential Decay (b) Polynomial Decay (c) Sliding Window Decay

FIG. 4.8. *Update Speed vs Decay Degree ($\epsilon = 0.05$, $C = 60$)*

We perform the experiments for the time decayed sum as well as the time decayed selectivity. Note that selectivity estimation generalizes the problems of estimating the *rank*, *ϕ-quantiles* and *frequent elements* (Theorem 3.27).

Results. Our main observations from the experiments are as follows. First, the actual space used by the sketch can be much smaller than the theoretically derived bounds, while the accuracy demand for estimation is still met. Next, the sketch can be updated quickly in an online fashion, allowing for high throughput data aggregation.

5.1. Experimental Setup. We implemented the sketch and the RangeSample algorithm [33] in C++, using gcc 3.4.6 as the compiler and making use of data structures from the standard template library (STL). The space usage is reported in terms of the number of nodes present in the sketch after the data stream is processed. The input stream is generated from the log of web request records collected on the 58th day of the 1998 World Cup [34], and has $32,355,332$ elements, of which $24,498,894$ are distinct. All experiments were run on a 2.8GHz Pentium Linux machine with 2GB memory.

Data Preparation. For repeatability, we present the transformation we performed on the original data set from the 1998 World Cup. Note that these transformations are not part of the sketch that we have designed, and are only used to create the experimental inputs. Each web request record r is a tuple: $\langle timestamp, clientID, objectID, size, method, status, type, server\rangle$. All the records are archived in [34] in the ascending order of the *timestamp*, which is the number of seconds since the Epoch. Our goal is to transform the set of records into a data stream which has asynchrony in the timestamps and has a reasonable percentage of duplicates.

STEP 1: Project each r to a stream element $e = (v, w, t, id)$. (1) $e.id = r.timestamp \mod 86400 + r.clientID \mod 100 + r.serverID \mod 100$. Note that "+" is the string concatenation, thus $id_{max} = 863,999,999$. The timestamp is taken modulo 86400 since all the data is collected from a single day. Binding $\langle r.timestamp, r.clientID, r.server\rangle$ together into $e.id$ results in the stream having a reasonable percentage of duplicates, because at a certain point of time, the number of web requests between a given pair of client and server is very likely to be one, or a number slightly larger than one. (2) $e.v = r.size \mod 10^9$. (3) $e.w = r.objectID$

mod 10^3, hence $w_{max} = 999$. (4) $e.t = r.timestamp \mod 86400$.

STEP 2: Make the duplicates consistent. Note that the duplicates from Step 1 may differ in either w or v. We sort the stream elements in ascending order of id (hence also in increasing order of t), then replace the duplicates with the first copy.

STEP 3: Create the asynchrony. We divide the stream into multiple substreams, such that the elements in each substream have the same *server*. Then we interleave the substreams into a new stream as follows. We remove the first element of a randomly selected non-empty substream and append it into the new stream, until all the substreams are empty.

STEP 4: Create the *processing time* of each stream element. Since w, t and the *processing time* determine the decayed weight of e when it is processed, every stream element needs to have the same *processing time* in a repetition of any experiment. The *processing time* of e is generated as follows: (1) $\Pr[delay = i] = \frac{1}{3}$, $i \in \{0, 1, 2\}$. (2) If the processing time of the previous element is larger than that of the current stream element, we assign the processing time of the previous element to the current element, as the processing time must be non-decreasing. Note that whenever we receive a query for the aggregate of interest, we assume the current clock time (query time) is the processing time of the most recently processed stream element.

5.2. Accuracy vs Space Usage. Recall that the theoretically derived sample size is $\frac{C}{\epsilon^2}$ for an ϵ-approximation (with probability $\geq \frac{2}{3}$) of the time decayed sum ($C = 60$, Theorem 3.11) and the time decayed selectivity ($C = 492$, Theorem 3.26). However, in the course of our experiments, we found that the desired accuracy could be achieved using much smaller values of C (and hence much smaller space) than the theoretical prediction.

Figure 4.2, 4.3, 4.4 and 4.5, shows the influence of C on the accuracy of estimations of the sum and the selectivity. In these experiments we set $\epsilon = 0.05, \alpha = 1$, $\beta = 0.01$ and $W = 200$ seconds. We use the following three predicates for selectivity estimation: (1) $P_1(v, w) = 1$, if $v/w \geq 2$; otherwise, 0. (2) $P_2(v, w) = 1$, if $v/w \geq 3$; otherwise, 0. (3) $P_3(v, w) = 1$, if $v/w \geq 4$; otherwise, 0.

With each time decay model and each value for C, we perform 10 experiments estimating the sum over the whole stream (Figure 4.2). Each dot in these figures represents an estimate for the sum. The x-axis of the dot is the value for C used in the experiment and the y-axis represents the relative error in the estimate for the sum. The lower bound and upper bound lines in each figure set up the boundaries between which the dots are the ϵ-approximations. Similarly, for each decay model, each value for C and each predicate, we perform 10 experiments estimating the selectivity over the whole stream (Figure 4.3, 4.4 and 4.5), whereas the y-axis of each dot is the additive error in the estimate for the selectivity.

Figure 4.2, 4.3, 4.4 and 4.5 first show that not surprisingly, a larger C yields more accurate estimators for both sum and selectivity. The second observation is that even a value as low as $C = 2$ is good enough to guarantee an ϵ-approximation of the sum with probability $\geq \frac{2}{3}$, whereas $C = 1$ is sufficient in the case of the selectivity (for the predicates we considered). The second observation gives a crucial indication that in the real applications of this sketch, the actual value for C can be much smaller than the theoretical predictions.

We next studied the influence of C on the sketch size using four different decay functions in Figure 4.6. Besides the exponential decay and polynomial decay, for which α, β are assigned the same values as in Figure 4.2, 4.3 and 4.4, we also study the size of the sketch using the sliding window decay with window size $W = 200$ seconds and $W = 25$ hours. Note that all the data in the experiments was collected within a day, therefore the sketch using the sliding window decay with $W = 25$ hours is a general sketch, which has enough information to answer time decayed sum or selectivity queries using any decay model (Section 4.2). Figure 4.6 shows that a smaller C can significantly reduce the sketch size, e.g., if $C = 2$, then the sketch size is about 10KB, whereas if $C = 20$, the sketch size is about 100KB. Figure 4.6 also shows that for the same value for C, the sliding window for $W = 25$ hours takes the most space, which is reasonable, since it can answer the broadest class of queries.

Overall, compared with the size of the input (over 32 million), the sketch size is significantly smaller. Note that the sketch size is independent of the input size, meaning even if the input is larger, the sketch will not be any larger, as long as the desired accuracy remains the same. Small sketch size is crucial for the sensor data aggregation scenario since the energy cost in the data transmission within the network can be significantly reduced by transmitting the sketches between nodes rather than the whole data.

5.3. Time Efficiency. In this section, we present experimental results for the time taken to update the sketch for different decay functions and parameter settings. We report the updating speed in terms of the number of stream elements processed per second. Our experiments demonstrate that overall, the sketch can be updated quickly (in the order of 10,000 updates per second).

Figure 4.7(a) shows the time (in seconds) taken to update the sketch for exponential decay, polynomial decay and sliding window decay. It shows that if $C = 60$, the sketch can handle about 15000 elements per second. If $C = 2$, the speed of updating is more than doubled, since a smaller C yields a smaller sketch (as shown in Figure 4.6), and smaller the sketch, faster are the operations on the sketch. Similarly, a higher accuracy demand (a smaller ϵ) slows down the sketch update (Figure 4.7(b)).

Both Figures 4.7(a) and 4.7(b) show that the sketch using polynomial decay has the highest time efficiency, whereas the sketch using the sliding window decay has the lowest time efficiency. This may come as a surprise, since exponential decay is often considered to be the "easiest" to handle, and polynomial decay is thought to be "harder". The reasons for our results are the parameter settings that we used for exponential and polynomial decay, and the distribution of the processing delays. In the experiments shown, we set $\alpha = 1.0$, causing a rather "fast" polynomial decay, and $\beta = 0.01$, causing a rather “slow” exponential decay. Of course, even with these settings, exponential decay will still cause the weights to decay “faster” than polynomial decay for very old elements, which are being processed long after they were generated. Due to the way we constructed our input, the processing delay of most stream elements were within 3 seconds. As a result, for most elements, when they are processed, their weight in polynomial decay was smaller than their weight in exponential decay, and their weight in sliding window decay was the largest. Since a smaller decayed weight implies an insertion into fewer samples, and the cost of computing the expiry time for a particular level is the same for all three decay models, polynomial decay resulted in the fastest processing time, while sliding window decay (with window size 200 seconds) led to the slowest processing time.

In general a sketch working with a decay function that decays “faster”, i.e., a larger value for α and β in polynomial decay and exponential decay respectively, or a smaller value for W in sliding window decay, has better time efficiency, because a “faster” decay function makes the weight of the element smaller, hence fewer insertions are performed on the sketch. This is shown in Figure 4.8(a) and 4.8(b), where for either exponential decay or polynomial decay, the time efficiency increases as the decay becomes faster. However, at the first glance, this is not the case for the sliding window decay displayed in Figure 4.8(c), and the update speed does not seem to change significantly with W. This is because in our experiments the ages of most elements at their processing time are no more than the smallest window size considered, 200 seconds, therefore the decayed weights of an element at its processing time using the sliding window decay of different window sizes ($W \in \{200, 400, 600, 800, 1000\}$) are the same (equal to the original weight).

6. Concluding Remarks. In this work, we have presented a powerful result. There exists a single sketch that allows duplicate-insensitive, distributed, and time-decayed computation of a variety of aggregates over asynchronous data streams. This sketch can accommodate any integral decay function, or any decomposable decay function, via the reduction to sliding window decay. For the class of decomposable decay functions, the decay function need not even be known a priori, and can be presented at query time.

We experimentally show that the actual space needed by the sketch can be significantly smaller than theoretical predictions, while still meeting the accuracy demands. Our experiments confirm that the sketch can be updated quickly in an online fashion, allowing for high throughput data aggregation.

REFERENCES

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

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

[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 *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 286–296, 2004.

[5] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 633–634, 2002.

[6] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *Proc. 22nd ACM Symp. on Principles of Database Systems (PODS)*, pages 234–243, June 2003.

[7] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *Proc. 3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 363–374, 2007.

[8] V. Braverman and R. Ostrovsky. Smooth histograms for sliding windows. In *FOCS*, pages 283–293, 2007.

[9] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *Proc. 30th ACM Symp. on the Theory of Computing*, pages 327–336, May 1998. Full version to appear in JCSS special issue for STOC'98.

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

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

[12] 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 *Proceedings of the 7th International Conference on Mobile Data Management (MDM)*, page 26, 2006.

[13] E. Cohen and H. Kaplan. Spatially-decaying aggregation over a network: model and algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 707–718, 2004.

[14] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. *Journal of Algorithms*, 59(1):19–36, 2006.

[15] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *20th International Conference on Data Engineering (ICDE)*, pages 449–460, 2004.

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

[17] G. Cormode, S. Tirthapura, and B. Xu. Time-decaying sketches for sensor data aggregation. In *Proc. 26th annual ACM symposium on Principles of distributed computing (PODC)*, 2007.

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

[19] J. Feigenbaum, S. Kannan, and J. Zhang. Computing diameter in the streaming and sliding-window models. *Algorithmica*, 41:25–41, 2005.

[20] P. Flajolet and G. N. Martin. Probabilistic counting. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 76–82, November 1983.

[21] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 281–291, 2001.

[22] P. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 63–72, 2002.

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

[24] P. Indyk and D. Woodruff. Tight lower bounds for the distinct elements problem. In *Proc. 44th IEEE Symp. on Foundations of Computer Science (FOCS)*, page 283, 2003.

[25] N. Kimura and S. Latifi. A survey on data compression in wireless sensor networks. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC)*, pages 8–13, 2005.

[26] T. Kopelowitz and E. Porat. Improved algorithms for polynomial-time decay and time-decay with additive error. *Theory of Computing Systems*, 42(3):349–365, 2008.

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

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

[29] 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.

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

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

[32] S. Nath, P. B. Gibbons, S. Seshan, and Z. Anderson. Synposis diffusion for robust aggregation in sensor networks. In *Proc. 2nd international conference on Embedded networked sensor systems*, pages 250–262, 2004.

[33] A. Pavan and S. Tirthapura. Range-efficient counting of distinct elements in a massive data stream. *SIAM Journal on Computing*, 37(2):359–379, 2007.

[34] Web requests for the 1998 world cup. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[35] B. Xu, S. Tirthapura, and C. Busch. Sketching asynchronous streams over a sliding window. *Distributed Computing*, 20(5):359–374, February 2008.

[36] L. Zhang and Y. Guan. Variance estimation over sliding windows. In *Proc. 26th ACM Symposium on Principles of Database Systems (PODS)*, pages 225–232, 2007.