Forward Decay: A Practical Time Decay Model for Streaming Systems

Graham Cormode [#], Vladislav Shkapenyuk [#], Divesh Srivastava [#], Bojian Xu [1,*]

[#]*AT&T Labs–Research, Florham Park NJ, USA*
{graham,vshkap,divesh}@research.att.com

[*] *Dept. of ECE, Iowa State University, Ames IA, USA*
bojianxu@iastate.edu

***Abstract*— Temporal data analysis in data warehouses and data streaming systems often uses time decay to reduce the importance of older tuples, without eliminating their influence, on the results of the analysis. While exponential time decay is commonly used in practice, other decay functions (e.g. polynomial decay) are not, even though they have been identified as useful. We argue that this is because the usual definitions of time decay are "backwards": the decayed weight of a tuple is based on its age, measured backward from the current time. Since this age is constantly changing, such decay is too complex and unwieldy for scalable implementation.**

In this paper, we propose a new class of "forward" decay functions based on measuring forward from a fixed point in time. We show that this model captures the more practical models already known, such as exponential decay and landmark windows, but also includes a wide class of other types of time decay. We provide efficient algorithms to compute a variety of aggregates and draw samples under forward decay, and show that these are easy to implement scalably. Further, we provide empirical evidence that these can be executed in a production data stream management system with little or no overhead compared to the undecayed computations. Our implementation required no extensions to the query language or the DSMS, demonstrating that forward decay represents a practical model of time decay for systems that deal with time-based data.

I. Introduction

"One today is worth two tomorrows"
— Benjamin Franklin

In processing data with timestamp information, it is common to downweight tuples which correspond to older data. Intuitively, this reflects the belief that the most recent data is most relevant for query answering, while older data is of less significance, and can be counted at a lower weight or ignored entirely. Notions of such "time decay" have been widely adopted across a broad class of systems, such as data warehouses, data streaming systems, sensor networks, and other distributed monitoring systems. Here, we focus on systems for managing data streams, although our analysis and observations apply broadly to all these settings.

Building robust systems for managing data streams is a challenging task, since typical streams (in application areas such as networks and financial data) arrive at very high rates and require immediate processing. Queries are typically continuous, meaning that the output of a query is itself a stream, which may be the input for subsequent querying. Systems must also cope with data quality issues: for example, there is no guarantee that tuples will be presented in timestamp order, and so techniques such as punctuations [36] and heartbeats [25] are used to avoid query blocking. A number of general purpose prototype streaming systems have been created, such as Stream [35], Aurora [38] and TelegraphCQ [8]; the current state-of-the-art deployed streaming systems (including GS [15] and Streambase [34]) are specialized for particular application domains (networking and financial).

Motivated by such applications, there has been a great deal of work on algorithms for efficiently answering streaming queries under time decay. Much of this focus has been on giving approximate answers to aggregate queries. However, within current production systems, the support for time decay is actually quite limited. We give our examples and evaluation using GS, a mature network stream processing system developed at AT&T [15]. This system allows a wide variety of queries to be posed in an SQL-like language, and has many hooks in it for extensibility: support for user defined operators (UDOPs) and user defined aggregate functions (UDAFs), which allow arbitrary (C/C++) code to be executed on selected tuples. This infrastructure has enabled approximate algorithms to be evaluated in the non-decayed case [10]. Yet support for time decay has so far been limited to a simple time-bucket approach: the query specifies a duration, such as the time in the granularity of minutes, and an answer is provided for each minute-wise time-bucket.

On closer inspection, it is clear that many of the approaches proposed so far for handling time decay do not scale well within streaming systems. Answering queries with a sliding window exactly requires buffering large quantities of tuples. While the approximate solutions, such as exponential histograms and its variants [17], [20], [12], improve the resources needed, they can still be of the order of megabytes of space per group and milliseconds of time per tuple to track complex holistic aggregates. But the motivating applications can typically only afford a few kilobytes of space per group in a query (since there can be tens of thousands of active groups) and microseconds per update, at best. So while these solutions

[1]Work done while visiting AT&T Labs–Research.

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.65

(surveyed in more detail in Section VII) have good asymptotic performance, they are not yet suitable for deployment in high throughput systems.

The complexity of existing algorithms for time decay arises because work so far has mostly concentrated on the case that we dub *backward decay*. That is, the weight of an item is computed based on its age, measuring *back* from the current time. This definition is motivated based on physical analogies: backward decay based on an exponential function resembles radioactive decay; with a polynomial function, it resembles the dispersion of (sound) energy. But implementing such decay is problematic, since an item's age changes as time elapses, making it necessary to maintain a lot of additional information to recompute the relative weights for the query.

Our Contributions. We propose a new class of decay functions which instead measures the age of an item *forwards* from an appropriate landmark point. Thus we call this class *forward decay*. It has the advantage that it can be much easier to compute with, since the "forward age" of an item (relative to the landmark) is fixed once it has been observed; its relative importance diminishes as newer items are seen, since their weights grow to dominate the older weights. It can also be motivated by physical analogies, again to radioactive decay, but also to (visual) perspective: the apparent height of objects is captured by a linear decay function, reaching zero at the horizon (the landmark).

We show several important properties of forward decay:

- Exponential decay is identical under both forward and backward decay models. The forward view of exponential decay helps to explain why this decay model is easier to compute; it also allows us to propose simple, effective algorithms for sampling under exponential decay, which strictly improve on the state of the art.
- For a large class of functions, specifically the monomials, forward decay guarantees a useful *relative decay property*, which is that the effective weight of an item is a function of its *relative age*: how far it falls along the interval between the landmark time and the current time. This is a natural and intuitive property that was not attainable under backward decay models.
- Forward decay captures and generalizes the existing notions of landmark windows.

Our analysis shows how forward decay can be computed using existing techniques for aggregates on weighted tuples in data streams. As a consequence, efficient and scalable algorithms follow immediately, with the same space and time bounds as their undecayed counterparts. Further, we implement these within the GS system, and compare to a selection of general techniques for backward decay. Simple aggregation such as count and sum is immediate, while holistic aggregates such as quantiles and heavy hitters require only appropriate UDAFs for the weighted versions of the aggregates. No extensions to the query language or changes to the system are needed. We observe that the forward decay solutions are practical for use in high speed systems, in contrast to the backward decay methods. In our experiments on live network streams, we observed that the forward decay approach could answer queries on multi-gigabit data without loss, while methods based on backward decay dropped many packets, and reached 100% CPU load.

Outline. We proceed as follows: In Section II we describe decay models and existing backward decay definitions, then in Section III we introduce our model of forward decay and study its properties. We show how to compute aggregates under forward decay in Section IV, and how to draw samples in Section V. Implementation issues are discussed in Section VI, related work in Section VII, and our experimental study is described in Section VIII.

II. Decay functions

We consider streams of input items (t_i, v_i), which describe item arrivals. We assume the i'th arrival has an associated time stamp t_i (this can be either given explicitly or be implicit as the arrival time). It may also have some associated value v_i: this can correspond to a description of the item (for example, the source-destination pair of a network packet), or a count associated with that item.

Definition 1: A *decay function* takes some information about the ith item, and returns a weight for this item. It can depend on a variety of properties of the item such as t_i, v_i as well as the current time t, but for brevity we will write it simply as $w(i,t)$, or just $w(i)$ when t is implicit. We define a function $w(i,t)$ to be a *decay function* if it satisfies the following properties:
1. $w(i,t) = 1$ when $t_i = t$ and $0 \leq w(i,t) \leq 1$ for all $t \geq t_i$.
2. w is monotone non-increasing as time increases: $t' \geq t \Rightarrow w(i,t') \leq w(i,t)$.

A. Backward Decay Functions

Prior work on time decay has focused on decay functions of a particular form: where the weight of an item can be written as a function of its *age*, a, where the age at time $t > t_i$ is simply $a = t - t_i$. We refer to decay of this form as *backward decay*, since we are always measuring *back* from the current time to the item's timestamp. More formally, we state:

Definition 2: A backward decay function is defined by a positive monotone non-increasing function $f()$ so that the weight of the ith item at time t is given by

$$w(i,t) = \frac{f(t-t_i)}{f(t-t)} = \frac{f(t-t_i)}{f(0)}$$

The denominator in the expression normalizes the weight, so that it obeys condition 1 of Definition 1. Some examples of the most popular decay functions are generated by picking f to be of a certain form, such as:

No decay. The trivial function $f(a) = 1$ for all ages a weights all ages equally. This means that the time-decayed model captures prior work on non-decayed aggregates.

Sliding Window. Given a "window size" parameter, W, the function $f_W(a) = 1$ for $a < W$ and $f_W(a) = 0$ for $a \geq W$

captures the common sliding window semantics—only items whose age is less than W are considered.

Exponential Decay. The class of functions $f(a) = \exp(-\lambda a)$ for $\lambda > 0$ has been used for many applications in the past. Part of its popularity stems from the ease with which it can be computed for sums and counts: given a new update, the exponentially decayed sum can be found by multiplying the previous sum by an appropriate amount, then adding on the weight of the new arrival. This backward decay function ensures that the time for f to drop by a constant fraction is the same, i.e. for a fixed delay A, the ratio $f(a)/f(A+a)$ is the same for all a.

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

Many other classes of backward decay are possible simply by choosing a different form for f, including super-exponential decays (e.g. $f(a) = \exp(-\lambda a^2)$) and sub-polynomial decays (e.g. $f(a) = (1+\ln(1+a))^{-1}$). It is easy to verify that all the above functions satisfy the requirements for decay functions (Definition 1).

There has been significant study of how to compute a variety of simple and complex aggregates under decay functions [9], [28], [12] (especially the special case of sliding window [4], [17], [26]). Typically their cost is high: the space and time required to apply decay can be many times the cost of computing the aggregate without decay. We survey these results in more detail in Section VII.

III. Forward Decay

The main challenge in implementing time decay computations under a backward decay function is that we must compute a function of the *age* of each item, relative to the current time, and this is constantly changing. To compute a simple decayed aggregate exactly, such as decayed sum, can require revisiting every input item to compute the contribution of that item (an exception is exponentially decayed sum and counts, which can be tracked in constant space due to properties of the decay function).

Instead we propose *Forward Decay* as a different model of decay satisfying Definition 1. The forward decay is computed on the amount of time between the arrival of an item and a fixed point L, known as the landmark. By convention, this landmark is some time earlier than all other items; we discuss how this landmark can be chosen below. Thus we are looking *forward* in time from the landmark to see the item, instead of looking *backward* from the current time.

Because we wish to weight more recent items more heavily than older ones, forward decay functions are based on monotone *non-decreasing* functions g. In order to normalize values given that the function value increases with time, we typically need to include a normalizing factor in terms of $g(t)$, the function of the current time. More formally,

Definition 3: Given a positive monotone non-decreasing function g, and a landmark time L, the *decayed weight* of an item with arrival time $t_i > L$ measured at time $t \geq t_i$ is given by

$$w(i,t) = \frac{g(t_i - L)}{g(t - L)}$$

This definition ensures that when $t = t_i$ the weight is 1 (condition 1 of Definition 1). As t increases, this weight never increases (due to the monotonicity of g) and remains in the range $[0, 1]$. Observe that scaling g by a constant has no effect on the value of the decayed weight.

Example 1: Consider the stream of (t_i, v_i) pairs

$$\mathcal{S} = \{(105, 4), (107, 8), (103, 3), (108, 6), (104, 4)\}$$

Let the landmark time $L = 100$, and set $g(n) = n^2$. Evaluated at $t = 110$, the decayed weights are respectively

$$\{0.25, 0.49, 0.09, 0.64, 0.16\}.$$

The shape of this decay function is plotted in Figure 1. ■

As with backward decay, the most natural choices of functions g fall into similar classes:

- No decay: $g(n) = 1$ for all n.
- Polynomial decay: $g(n) = n^\beta$ for some parameter $\beta > 0$.
- Exponential decay: $g(n) = \exp(\alpha n)$ for parameter $\alpha > 0$.
- Landmark Window: $g(n) = 1$ for $n > 0$, and 0 otherwise.

We discuss the properties of each of these classes of forward decay in turn.

A. Exponential Decay

We observe that forward exponential decay coincides exactly with backward exponential decay. Formally, consider item i which arrives at time t_i. Under backward decay and the function $f(a) = \exp(-\alpha(a))$, its decayed weight is $w(i,t) = \exp(-\alpha(t - t_i))$. Under forward decay, its decayed weight under the function $g(n) = \exp(\alpha(n))$ is

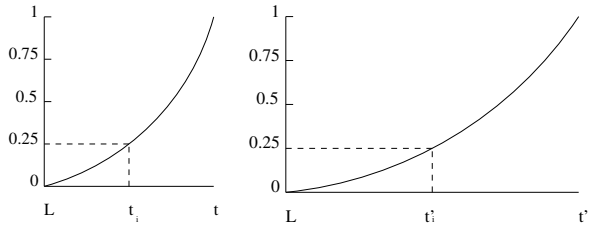
$$\begin{aligned}\frac{g(t_i - L)}{g(t - L)} &= \frac{\exp(\alpha(t_i - L))}{\exp(\alpha(t - L))} \\ &= \exp(\alpha t_i - \alpha L - \alpha t + \alpha L) \\ &= \exp(-\alpha(t - t_i)) = w(i,t)\end{aligned}$$

i.e. the two definitions precisely coincide. This is not the case for other classes of decay such as backward polynomial decay.

This observation motivates our study of forward decay, since it shows forward decay contains an important existing class of functions that have been widely studied and adopted. But more than this, viewing exponential decay from the forward decay perspective allows us to propose effective new algorithms for problems such as sampling (Section V).

B. Polynomial Decay

In general, one can specify arbitrary polynomial decay functions of the form $g(n) = \sum_j \gamma_j n^j$ for some set of γ_js. But the most natural polynomials to use are monomials, $g(n) = n^\beta$ for some exponent β. Under such decay functions, the decayed weights obey an important *relative decay* property.



(a) Weight at time t (b) Weight at later time t'

Fig. 1. Relative decay property for forward decay on $g(n) = n^2$

Definition 4: A system for determining decayed weights is said to have the *relative decay* property if, for any time t after a landmark time L, the weight for items with time stamp $\gamma t + (1-\gamma)L$ is the same.

In other words, if the weight assigned to an item depends only on where it falls as a fraction in the window defined by L and t, then it is relative decay. So for instance, the item arriving half way between L and t is assigned the same weight, as t increases. This should be an intuitive property: it asks that the weight assigned to an item is a function of its *relative age*, that is, its age as a fraction of the total time period observed. However, backward decay is only concerned with *absolute age*, and so gives no guarantee of relative decay.

Lemma 1: Forward decay based on a monomial function $g(n) = n^\beta$ satisfies the relative decay property.

Proof: Let $g(n) = n^\beta$. The weight for an item with arrival time $t_i = \gamma t + (1-\gamma)L$ evaluated at time t is given by

$$w(i,t) = \frac{g(t_i - L)}{g(t-L)} = \frac{g(\gamma(t-L))}{g(t-L)} = \frac{(\gamma(t-L))^\beta}{(t-L)^\beta} = \gamma^\beta$$

■

This is illustrated in Figure 1 with $g(n) = n^2$: at time t, in Figure 1(a), item t_i chosen to fall half-way between L and t has weight 0.25. This is true for any time t', as shown in Figure 1(b), where t_i' (also chosen to fall midway between L and t') has the same weight as before.

Landmark Choice. This observation is helpful in determining a meaningful landmark L to choose for forward decay: because of the relative decay property, it makes sense to set the landmark time to the start time (or just before) of the query in question. Then items with the same relative time within the span of timestamps associated with the query have the same decayed weight. From now on, we assume that the default for L for a given query is (a lower bound on) the smallest timestamp in the stream. For example, when timestamps are allocated as the system time at which the tuple is observed, we set L to be the time when the query was issued.

C. Landmark windows

Lastly, we observe that the natural equivalent of (backward) sliding window is the *Landmark window*, given by the forward decay function that assigns weight 1 to all items with timestamp greater than landmark L [21]. The window is said to "close" when the query terminates—perhaps based on seeing a certain number of tuples, or after a certain time has elapsed. This model has been implicitly adopted by many systems, since it is trivial to implement (just do regular aggregation until the window closes). Here, we give a foundation for this model by viewing it as a (simple) instance of forward decay.

IV. Aggregate Computation under Forward Decayed Models

A decay function in either the forward or backward setting assigns a weight to each item in the input (and the value of this weight can vary over time). Aggregate computations over such data must now use these weights to scale the contribution of each item. In most cases, this leads to a natural weighted generalization of the aggregate. We next work through choices of aggregates, and show their weighted generalization. We then discuss how to implement exact or approximate computation of these aggregates over n tuples assuming forward decay based on a function g and a landmark time L.

A. Count, Sum and Average

The three basic aggregates of Count, Sum and Average are straightforward to generalize under forward decay:

Definition 5 (Count, Sum and Average): The decayed count, C, is the sum of decayed weights of stream items

$$C = \sum_{i=1}^{n} (g(t_i - L)/g(t-L)).$$

The decayed sum, S, takes an additional value v_i for each item i, and sums the weighted values:

$$S = \sum_{i=1}^{n} (g(t_i - L)v_i/g(t-L)).$$

The decayed average, A, is the ratio of decayed sum to decayed count, so

$$A = S/C = (\sum_i g(t_i - L)v_i)/(\sum_i g(t_i - L)).$$

Example 2: Take the same example stream given in example 1. Then we have

$$\begin{aligned} C =& 0.25 + 0.49 + 0.09 + 0.64 + 0.16 = 1.63 \\ S =& 0.25 \cdot 4 + 0.49 \cdot 8 + 0.09 \cdot 3 + 0.64 \cdot 6 + 0.16 \cdot 4 = 9.67 \\ A =& S/C = 5.93 \end{aligned}$$

■

Observe that we can write $S = \frac{1}{g(t-L)}(\sum_i g(t_i - L)v_i)$. This can be computed by maintaining the value of $\sum_i g(t_i - L)v_i$, and scaling by the value of $g(t-L)$ only when needed for output. C can be maintained in the same fashion, and A is given by the ratio of these two values. Note that the value of the average under this definition does not vary as the current time t increases: this is because the average gives an average of the input values, weighted towards the more recent ones. But, for instance, if all items have the same value v, then their average should be v no matter when the query is executed, which is obeyed by our definition.

Other simple numeric quantities can be computed similarly. For example, the decayed variance V (interpreting weights as

probabilities) can be written in terms of the decayed sum of squared values, $V = \sum_i g(t_i - L)v_i^2/C - A^2$. More generally, the decayed version of any summation of an algebraic expression of tuple values (i.e. one based on standard arithmetic operations such as addition, multiplication and exponentiation) is found by computing the value of the expression on tuple t_i, multiplying by $g(t_i - L)$. The final result is found by scaling the sum by $g(t - L)$ at query time t. Thus:

Theorem 1: Any summation of an arithmetic operation on tuples that can be computed in constant space without decay can also be computed in constant space under any forward decay function.

This has immediate implications for any high-performance streaming system: simple algebraic quantities can be computed under any forward decay function using existing arithmetic support. This can be specified directly in the query by spelling out the function to create the weights, or by adding some simple syntactic sugar to achieve the same effect. For example, within the GS query language (GSQL), we can express a decayed count query under quadratic decay as:

```
select tb, destIP, destPort,
sum(len*(time % 60)*(time % 60))/3600 from TCP
group by time/60 as tb, destIP, destPort
```

Here, the query finds the (decayed) sum of lengths of packets per unique destination (port, address) pair, within a window constrained to 60 seconds (hence the scaling by $60^2 = 3600$). Since it is expressed entirely in the high-level query language, the optimizer can decide how to execute it, find shared subexpressions etc.

These results are in contrast to backward decay functions: prior work has shown approximation algorithms for sum and count with $1+\epsilon$ relative error for any backward decay function, but requiring a blow up in space by an $O(\frac{1}{\epsilon}\log n)$ factor.

B. Min and Max

For Min (respectively, Max), we want to find the tuple which has the smallest (largest) associated *decayed* value. Under backward decay functions, this is a challenging task, since the changing value of the decay function over time causes the value of the Min (Max) to vary over time. In contrast, applying the definition to forward decay generates the following definition:

Definition 6 (Min and Max): The decayed minimum value MIN is defined as

$$\begin{aligned} MIN &= \min(g(t_i - L)v_i/g(t - L)) \\ &= \frac{1}{g(t-L)} \min_i g(t_i - L)v_i \end{aligned}$$

and the decayed maximum value MAX is defined as

$$\begin{aligned} MAX &= \max(g(t_i - L)v_i/g(t - L)) \\ &= \frac{1}{g(t-L)} \max_i g(t_i - L)v_i. \end{aligned}$$

Observe that in both cases it suffices to compute the smallest (greatest) value of $g(t_i - L)v_i$ seen so far. For MAX, when a new (t_i, v_i) pair is observed, compute the corresponding value of $g(t_i - L)v_i$, and retain the item if it exceeds the largest value seen so far. As for algebraic aggregates, this is easily computed within a streaming system as a simple extension of the undecayed aggregate. In contrast, this problem is provably hard to solve in small space under backward decay, since in the sliding window case we can force the algorithm to "remember" the entire contents of the window.

C. Heavy Hitters and Quantiles

For holistic aggregates such as Heavy Hitters and Quantiles, it is more complicated to find the answer to queries. However, we will show approximate solutions to the problem with forward decay which have the same asymptotic costs as their undecayed equivalents. Meanwhile, for backward decay, methods take at least a logarithmic factor more space (Section VII).

Approximate Heavy Hitters. First, we formally define the heavy hitters problem:

Definition 7 (Heavy hitters under forward decay): For each item in the input, v, its decayed count is given by $d_v = \sum_{v_i=v} g(t_i - L)/g(t - L)$. Given a threshold value ϕ, the ϕ heavy-hitters are all items v satisfying $d_v \geq \phi C$.

Example 3: Consider the example stream given in Example 1. We have $C = 1.63$, and

$$d_3 = 0.09, d_4 = 0.16 + 0.25 = 0.41, d_6 = 0.64, d_8 = 0.49$$

Setting $\phi = 0.2$, the ϕ heavy hitters are items 4, 6, and 8, since their decayed counts exceed $1.63 * 0.2 = 0.326$. ■

Observe, as in heavy hitters without decay, that $\sum_{i=1}^{n} d_i = C$, where C is the (decayed) count given by Definition 5. The (decayed) heavy hitters are those items whose (decayed) count is at least a ϕ fraction of the total (decayed) count. Efficiently computing the heavy hitters over a stream of arrivals is a challenging problem that has attracted much study even in the unweighted, undecayed case. The difficulty comes from trying to keep track of sufficient information while using much fewer resources than explicitly tracking information about each distinct item. Here, efficient approximate solutions are known. Given a parameter ϵ, these approximate solutions may give an error in the estimated (decayed) count of items of at most ϵ times the sum of all (decayed) counts.

Theorem 2: Given an error bound ϵ, we find all items with $d_v \geq \phi C$, and report no items with $d_v < (\phi - \epsilon)C$ under the forward decay model using space $O(1/\epsilon)$ counters, and processing each update in time $O(\log 1/\epsilon)$.

Proof: Observe that we can rewrite the requirement as

$$\begin{aligned} d_v g(t - L) &\geq \phi C g(t - L) \\ \text{or equivalently } \sum_{v_i=v} g(t_i - L) &\geq \phi \sum_i g(t_i - L). \end{aligned}$$

In other words, we can treat this as an instance of a weighted heavy hitters problem, where the weight of each item is set on arrival as $g(t_i - L)$. Importantly, these weights do not change over time.

We can use the SpaceSaving algorithm proposed by Metwally *et al.* [29]. As analyzed in [11], this algorithm naturally

extends to weighted updates. We omit full details of the proof for brevity; the proof in [11] is in the context of exponentially decayed updates, but holds for arbitrarily weighted updates. The running time and resources needed are the same as the original SpaceSaving algorithm, which can be implemented in the given bounds. ■

Approximate Quantiles. The quantiles of a distribution generalize the median, so that the ϕ quantile is that item which dominates a ϕ fraction of the other items. As with heavy hitters, a natural weighted generalization can be used over time-decayed weights: we now search for an item that dominates a ϕ fraction of the decayed weights. Formally,

Definition 8 (Quantiles under forward decay): For each item v, its decayed rank is computed as $r_v = \sum_{v_i \leq v} g(t_i - L)/g(t - L)$. Given a query value ϕ, the ϕ quantile is the smallest item v satisfying $r_v \geq \phi C$.

Again, exact computation of quantiles can be costly over large data sets, since it requires keeping information about the whole input. Instead, approximate quantiles tolerate additive error ϵ in the rank (relative to the maximum rank). We will assume that the items are drawn from an integer domain of size U, i.e. each $v_i \in [1, U]$. Then:

Theorem 3: Given an error bound ϵ, we find decayed ϕ-quantiles under forward decay using space $O(\frac{1}{\epsilon} \log U)$ counters, and processing each update in time $O(\log \log U)$.

Proof: Similarly to heavy hitters, we can factor out the $g(t - L)$ term, so that we reduce the problem to find the smallest item i such that $\sum_{v_i \leq v} g(t_i - L) \geq \phi \sum_i g(t_i - L)$. This is a weighted quantiles problem defined over the (static) weights $g(t_i - L)$. We can now make use of solutions to weighted quantiles problems. The q-digest data structure [33], [11] naturally handles weighted updates and answers the approximate quantiles problem with the bounds given in the statement of the theorem. ■

This approach applies to other holistic aggregate computations over data streams (e.g. clustering and other geometric properties [22], [24]): factor out the $g(t-L)$ term and track the input using weights $g(t_i - L)$. We suppress further examples that fit this pattern for brevity.

D. Count Distinct

Aggregates with `distinct` keywords, such as Count Distinct, are a little more complicated to handle. It is not immediately obvious how to extend the count distinct aggregate to the weighted scenario. We proceed by analogy with the undecayed case: there, we can view the process as computing a single weight for each distinct item and summing these weights to get the overall aggregate. In the undecayed case, the weight for each distinct item present in the input is always 1. So for the weighted (time decayed) case, the natural generalization is to compute some function of the weights of each distinct item and sum these. For time decay, the weight of an item begins at 1 and decays towards 0, so we choose to define the representative weight of a set of items as the maximum of their current weights. This generalizes the unweighted case, which can be thought of taking the max of the set of the (all 1) values attached to each distinct item. More formally,

Definition 9 (Count Distinct under forward decay): The distinct count D of a set of items under forward decay is

$$D = \sum_v \max_{v_i = v} g(t_i - L)/g(t - L).$$

This definition seems to be justified, since it can be approximated using techniques based on careful combinations of unweighted count distinct summaries.

Theorem 4: Given a desired error bound ϵ, we can approximate D under the forward decay model within relative error $(1 \pm \epsilon)$ using space $\tilde{O}(\frac{1}{\epsilon^2})$.

Proof: We write distinct count under forward decay as

$$\frac{1}{g(t - L)} \sum_v \max_{v_i = v} g(t_i - L)$$

and so focus our effort on estimating the quantity $\sum_v \max_{v_i = v} g(t_i - L)$, which does not depend on the query time t. As before, $g(t_i - L)$ can be computed on arrival of the item, and does not vary with time. So we can write the weight of item i as $w(i, t) = w_i = g(t_i - L)$, and the desired quantity is $\sum_v \max_{v_i = v} w_i$. This now corresponds exactly to the "dominance norm" defined in [13]. The most efficient method to approximate this quantity is due to Pavan and Tirthapura [31], which generalizes techniques for counting the number of distinct items. Applied to our problem, the time cost is $\tilde{O}(\frac{1}{\epsilon^2})$ (with $\tilde{O}$ notation suppressing polynomial factors in $\log n$ and $\log \epsilon$). Each update takes time $\tilde{O}(1)$ time. The result is correct up to relative error $1 \pm \epsilon$ with high probability. ■

V. Sampling Under Forward Decay

The aggregate computations discussed in the previous section are each somewhat specific to a particular goal: finding heavy hitters, quantiles, and other pre-defined aggregates. It is also useful to generate generic summaries of large data, on which ad-hoc analysis can be performed after the data has been observed. The canonical example of such a summary is the uniform random sample: given a large enough sample, many aggregates can be accurately estimated by evaluating them on the sample. We discuss various techniques for sampling from data with weights determined by forward decay functions.

A. Sampling With Replacement

In sampling with replacement, we aim to draw samples from the population so that in each drawing, the probability of picking a particular item is the same. For the unweighted case, a single sample is found by the simple procedure of independently retaining the i'th item in the stream (and replacing the current sampled item) with probability $1/i$. Under forward decay, the probability of sampling item i should be

$$\frac{w(i, t)}{\sum_{i=1}^n w(i, t)} = \frac{g(t_i - L)}{\sum_{i=1}^n g(t_i - L)}$$

Theorem 5: We can draw a sample with replacement under forward decay in constant space, and constant time per tuple.

Proof: A simple generalization of unweighted version suffices to draw a sample according to this definition. Let $W_i = \sum_{j=1}^{i} g(t_j - L)$ denote the sum of the weights observed so far in the stream, up to and including item i. We choose to retain the ith item as the sampled item with probability $g(t_i - L)/W_i$. The probability that the ith item is chosen as the final sample is given by

$$\frac{g(t_i - L)}{W_i} \prod_{j=i+1}^{n} \left(1 - \frac{g(t_j - L)}{W_j}\right) = \frac{g(t_i - L)}{W_i} \prod_{j=i+1}^{n} \frac{W_{j-1}}{W_j} = \frac{g(t_i - L)}{W_n}$$

■

For a sample of size s, we repeat this procedure s times in parallel with different random choices in each repetition. As in Reservoir Sampling [37], the procedure can be accelerated by using an appropriate random distribution to determine the total weight of subsequent items to skip over.

B. Sampling Without Replacement

A disadvantage of sampling weighted items with replacement is that an item with heavy weight can be picked multiple times within the sampled set, which reveals less about the input. This is a particular problem when applying exponential decay, when the weights of a few most recent items can dwarf all others. There are many formulations of weighted sampling without replacement [30]. Here, we outline two approaches that work naturally for forward decay. Both are based on the observation that, since sampling should be invariant to the global scaling of weights, we can work directly with $g(t_i - L)$ as the weight of the ith item.

Weighted Reservoir Sampling. In weighted reservoir sampling (WRS), a fixed sized sample (reservoir) is maintained online over a stream. The algorithm of Efraimidis and Spirakis [19] draws a sample of size k without replacement, with same probability distribution as the following (offline) procedure: At each step i, $1 \leq i \leq k$, select an element from those that were unselected at previous steps. The probability of selecting each element at step i is equal to the element's weight divided by the total weights of items not selected before step i.

The (online) algorithm in [19] generates a "key" $p_i = u_i^{1/w_i}$ for the ith tuple, where w_i is the weight and u_i is drawn randomly from $[0 \ldots 1]$. The sample is the set of k items with the k largest key values. Since we can factor out $g(t-L)$ in forward decay and this does not affect the sampling probability for each element, we can set the weight of each tuple $w_i = g(t_i - L)$, and obtain a sample according to the weights in the forward decay model.

Priority Sampling. Priority sampling due to Alon *et al.* [3] also generates a sample of size k, with a similar procedure: now, the priority q_i is defined as w_i/u_i (where u_i is again uniform from $[0 \ldots 1]$), and the algorithm retains the k items with highest priorities. Such a sample can be used to give an unbiased estimator for any selection query. The variance of this estimator is proved to be near-optimal. For similar reasons, priority sampling can also be used over the streams with any decay function within the forward decay model.

Theorem 6: We can maintain a weight based reservoir of stream elements under the WRS or priority sampling models for any decay functions in the forward decay model using space $O(k)$ and time $O(\log k)$ to process each element.

The time bounds for the theorem follow by keeping the keys/priorities in a priority queue of size k. To our knowledge, there is no way to draw such samples over a stream for general backward decay functions without blowing up the space considerably greater than k.

C. Sampling Under Exponential Decay

The special case of drawing a sample under exponential decay has been posed previously, and a partial solution given for the case when the time stamps are sequential integers [2]. By using the forward decay view, we are able to provide a solution for arbitrary arrival times, using space proportional to the desired sample size.

Corollary 1: We can draw a sample of size k with weights based on exponential decay in the backward decay model using only $O(k)$ space.

The corollary follows immediately from the algorithms in Section V-B, and the fact shown in Section III-A that forward and backward exponential decay coincide. This strictly improves previously known solutions, and is quite simple, relying only on the ability to draw a weighted sample. This observation was possible by viewing the problem through the lens of forward decay; it appeared much more complex when viewed as a backward decay problem.

VI. Implementing Forward Decay

A. Numerical issues

A common feature of the above techniques—indeed, the key technique that allows us to track the decayed weights efficiently—is that they maintain counts and other quantities based on $g(t_i - L)$, and only scale by $g(t - L)$ at query time. But while $g(t_i - L)/g(t - L)$ is guaranteed to lie between zero and one, the intermediate values of $g(t_i - L)$ could become very large. For polynomial functions, these values should not grow too large, and should be effectively represented in practice by floating point values without loss of precision. For exponential functions, these values could grow quite large as new values of $(t_i - L)$ become large, and potentially exceed the capacity of common floating point types. However, since the values stored by the algorithms are linear combinations of g values (scaled sums), they can be rescaled relative to a new landmark. That is, by the analysis of exponential decay in Section III-A, the choice of L does not affect the final result. We can therefore multiply each value based on L by a factor of $\exp(-\alpha(L' - L))$, and obtain the correct value as if we had instead computed relative to a new landmark L' (and then use this new L' at query time). This can be done with a linear pass over whatever data structure is being used.

B. Out-of-order and Distributed arrivals

It has recently been noted that many streams in practical applications do not arrive in exactly sorted order: delays or merging multiple streams can result in "late" arrivals. Under backward decay, this can require significant effort to accommodate [6], [12]. But for our forward decay methods, it is quite straightforward to accommodate, since nowhere do any of our proposed algorithms rely on items arriving in increasing order of timestamps. The only caveat is that we should ensure that queries are posed with time values t that are at least as big as the largest timestamp t_i observed so far—otherwise some decayed weights could exceed 1. Alternately, if we allow items whose time stamps are "in the future" relative to the query time parameter t, then one can pose historical queries in the forward decay model.

Similarly, we have phrased the discussion so far in terms of a single, centralized system. But the current trend is towards distributed, parallel systems (or within a single, multi-core, CPU). We comment that the definition of forward decay naturally extends to this model, and that all the techniques for aggregate computation and sampling discussed apply naturally to this scenario. In particular, given the data structures computed at each centralized site for the same decay function and landmark, they can easily be merged to form a data structure summarizing the union of the inputs. These details are mostly immediate from the definitions of the algorithms.

VII. Related Work on Time Decay

Related work on computing aggregates and samples with time decay has focused on two cases: sliding window decay, and other decay functions.

Sliding Window. The notion of a sliding window is a natural one when processing a stream of updates: since there are too many tuples to store (especially when processing joins), simply drop the oldest tuples. This simple definition holds much complexity, and has led to numerous papers and theses on processing this definition (see [21] and references therein). Various models have been proposed for the semantics of sliding windows. The Aurora system [7] defines sliding windows, which can overlap; tumbling windows, which have no overlaps; and latched windows, which are tumbling with preserved internal states. Li *et al.* [27] propose an approach based on panes: each window is divided into panes consisting of multiple tuples, so that each "slide" drops the oldest pane. GS typically provides tumbling window semantics by allowing queries to be based on "time-buckets" [16].

However, evaluating aggregate queries over sliding windows—even simple queries based on sum and count—can require a lot of state to be maintained, since tuples must be stored until they expire to correctly compute their effect on the aggregate. Consequently, there has been much research on approximate computation of aggregates under sliding windows using much smaller space resources. The earliest work focused on tracking sums and counts: both Exponential Histograms (EH) [17] and Deterministic Waves [20] answer these queries on a window of size N with relative error ϵ by keeping a careful arrangement of $O(\frac{1}{\epsilon}\log \epsilon N)$ counts and timestamps. They can extend to more complex aggregates by replacing their internal counts with other data structures such as sketches, but this causes the space to blow up by further multiples of $\frac{1}{\epsilon}$ and $\log N$.

For more complex holistic aggregates, such as quantiles and frequent items, Arasu and Manku proposed a generic approach with cost only a $\log \frac{1}{\epsilon} \log N$ factor larger than the unwindowed approximate algorithms [4]. Lee and Ting [26] reduce the space for frequent items for a fixed size window to $O(\frac{1}{\epsilon})$, the same as the unwindowed case. There has also been recent interest in handling cases where tuples with timestamps do not arrive in timestamp order: results have been shown for sums and counts [6], sampling [14] and quantiles and heavy hitters [12]. This flexibility comes at a cost: the bounds are further logarithmic factors more expensive than their ordered counterparts. Likewise, methods for sampling from a sliding window require space logarithmically (in the number of tuples in the window) larger than the desired sample size [5].

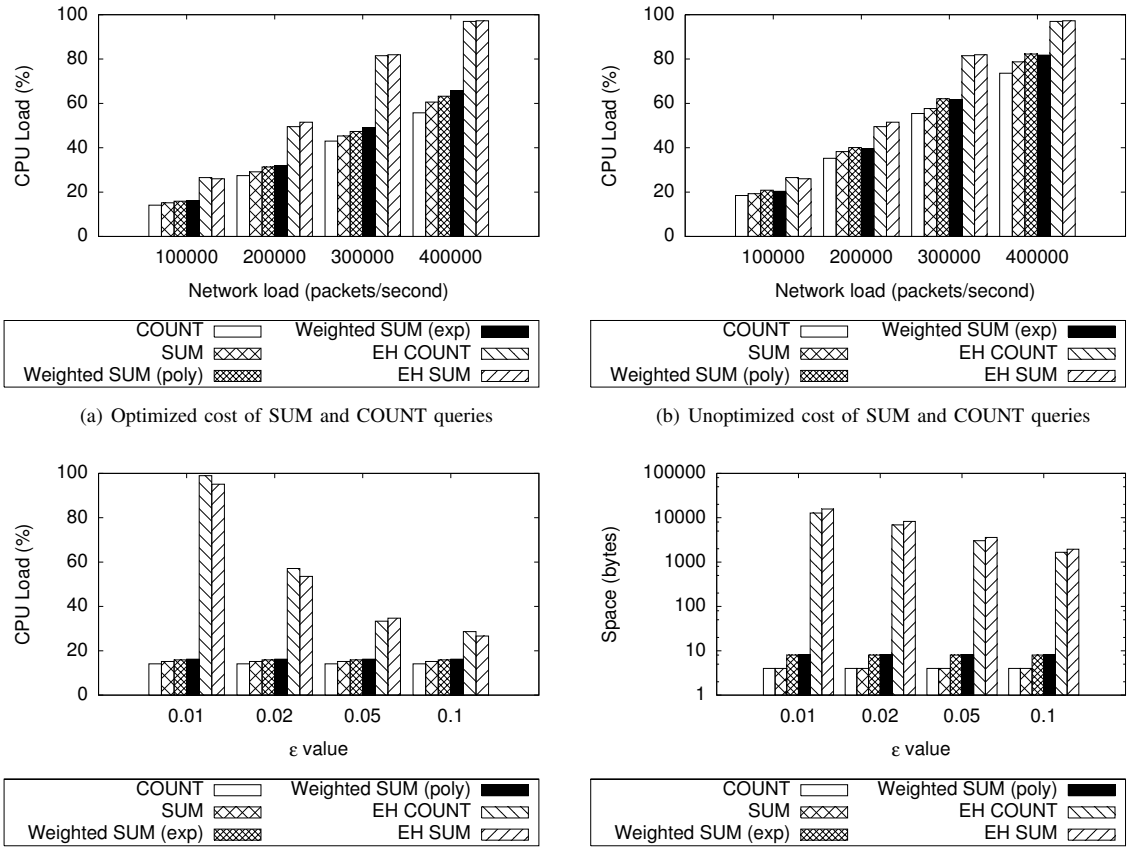
Other decay functions: exponential and polynomial decay. Among other decay functions, exponential decay is most popular, since a regular counter can be replaced with an exponentially decayed counter without increasing the (asymptotic) space cost. More recently, there has been interest in extending to aggregates beyond sums and counts, including sampling under exponential decay [2], and quantiles and heavy hitters [11], which obtain the same space bounds as the undecayed case. We explain this by our model, where forward and backward models of decay coincide for exponential decay.

For backward decay with other functions, such as a polynomial, the space cost is typically (much) higher. Cohen and Strauss introduced a variety of techniques for tracking sums and counts under backward decay [9], with cost $O(\frac{1}{\epsilon}\log N)$. This was extended to sampling and aggregate computation [14], [12], with similar blow-ups of $\mathrm{poly}(\frac{1}{\epsilon}, \log N)$ over the undecayed version. Our main results show that, in the different model of forward decay, all computations can be done in the same asymptotic resources as for undecayed aggregates.

VIII. Experimental Evaluation

In this section we present the results of experimental evaluation of several aggregate and sampling streaming algorithms under forward and backward decay models.

Experimental Set-Up and Environment. All the experiments were done in the context of the GS streaming database [15]. For simple aggregate queries (sum and count), we could write these using the built-in GSQL aggregate functions, `count()` and `sum()`. We compared the cost of these to that for Exponential Histograms (EH) [17], with variations for both sum and count. This makes for an interesting comparison, since, following the analysis of Cohen and Strauss [9], the EH is capable of approximating sum and count under any decay function (forward or backward) specified at query time: we can rewrite the decayed sum (resp. count) query as a sum of



(a) Optimized cost of SUM and COUNT queries

(b) Unoptimized cost of SUM and COUNT queries

(c) CPU load as ϵ is varied for EH

(d) Space usage as ϵ is varied for EH

Fig. 2. Experiments on Count queries under time decay

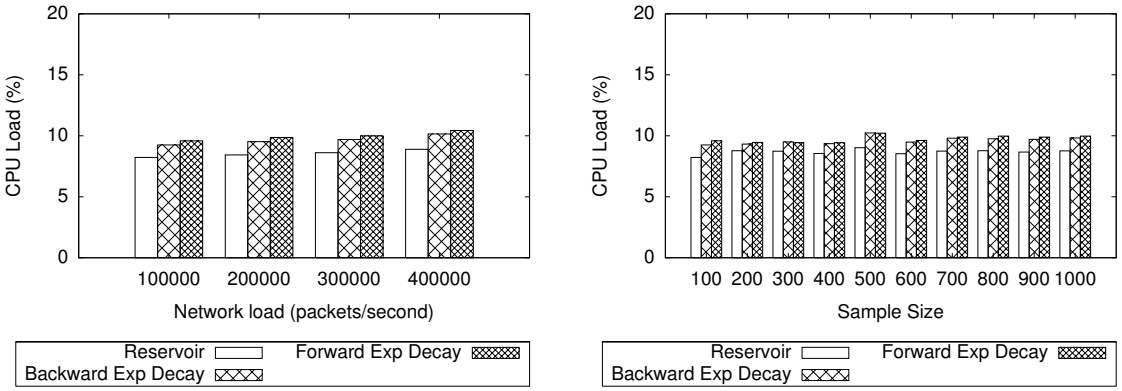
multiple scaled sliding window sum (count) queries, each of which can be answered approximately by the same EH data structure. So we can compare the cost of exactly computing the forward decay query to the best previous method, which would approximate it. We also compare against the baseline of directly computing the sum and count of the data, without adjusting for time decay.

For sampling, we performed a similar comparison against three classes of decay: no decay, forward decay, and backward decay. We used the traditional reservoir sampling approach to draw an unweighted sample [37], and compared the cost of this to priority sampling being supplied with exponentially increasing weights [3] and our implementation of Aggarwal's method for sampling under exponential decay [2]. For the backward decay, all weighting is internal to the UDAF implementing the decay, while for priority sampling, the UDAF implements standard priority sampling and the query generates the weights based on timestamps to feed in.

We also implemented weighted heavy hitters through the UDAF mechanism, using C code for the weighted version of the SpaceSaving algorithm discussed in Section IV-C[2]. Here, we compared to a method for answering sliding window heavy hitter queries [12]. As in the sum and count case, it can be shown that the results of multiple sliding window queries can be combined to form the answer to an arbitrary (forward or backward) decayed heavy hitter query. So again, we are comparing our techniques to approximate aggregate queries under decay with the best known previous method that could be used to accomplish it. We contrast both these decayed measures to the undecayed computation of heavy hitters, where we can use a version of the SpaceSaving algorithm that is optimized for unweighted (unary) updates.

All the experiments were conducted on live high-speed network traffic. We used two-CPU, dual-core 3.0Ghz Intel Xeon server with 4Gbytes of RAM running Linux 2.4.21, however only one core was used to run the code. In the course of the experiments the volume of observed network traffic was approximately 400,000 packet/sec (about 1.8 Gbit/sec). We could vary the effective stream rate presented to the system by adjusting the flow sampling rate performed in hardware on the network interface card.

[2]Our code is based on the routines at `http://www.research.att.com/~marioh/frequent-items`.



(a) Sampling as traffic load varies

(b) Sampling as sample size varies

Fig. 3. Experiments on Sampling Queries under time decay

A. Experimental Results.

Count and Sum Aggregates. These queries computed a summary (count or sum) of the traffic (presented as packets) sent to distinct TCP servers every minute. The undecayed query is expressed in GSQL as:

```
select tb, destIP, destPort, count(*)
from TCP
group by time/60 as tb, destIP, destPort
```

We compared the performance of sum and count queries with their weighted (backward and forward) counterparts. The results are shown in Figure 2. Figure 2(a) shows the effect as we varied the stream rate from 100,00 packets/sec to 400,000 packets/sec and observed the total CPU load. This shows the cost of forward-decayed aggregates with quadratic ("poly") and exponential decay ("exp") is a little higher than processing without decay, while supporting backward decay via exponential histograms (with parameter $\epsilon = 0.1$) has appreciably higher cost, and nearly saturates the system under high traffic load. For undecayed and forward-decayed aggregates the GS system can optimize the query over the system's two-level architecture. More precisely, the system splits the query into a low-level part performing partial aggregation using fixed-size hash-table and a super-aggregation query combining partial results. Our UDAFs were written to run at the high-level only. Figure 2(b) shows our effort to remove this advantage for the same queries by disabling this aggregate splitting in the system. However, there is still an appreciable cost of backward decay over forward decay.

This benefit becomes more pronounced as we vary the accuracy parameter ϵ of the exponential histograms. Recall that exponential histograms give an answer that is approximate to within relative error $1+\epsilon$, while the other queries are computed exactly. For the same queries as before, we decreased ϵ down to 0.01, while the stream data rate was set to 100,000 packets/second (Figure 2(c)). The throughput of undecayed and forward decayed aggregates does not alter, since they do not depend on ϵ. At $\epsilon = 0.01$, the backward decayed algorithms approach 100% CPU utilization and drop tuples.

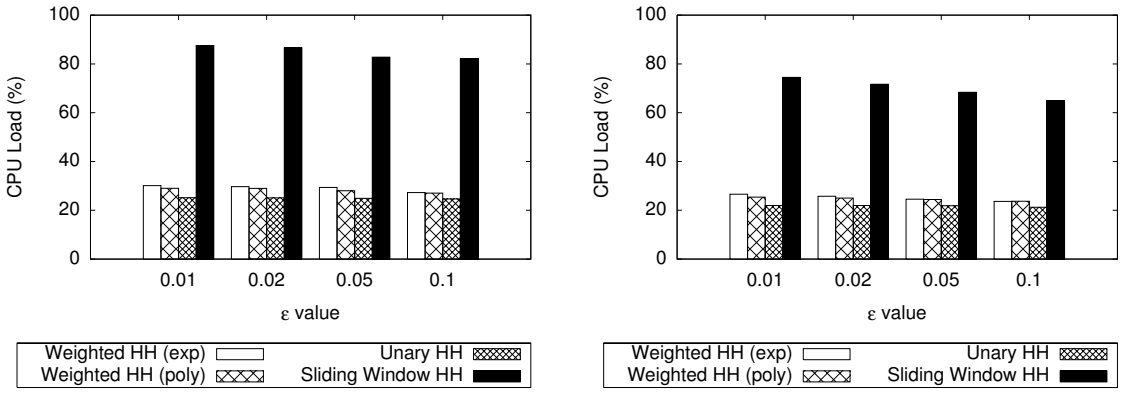
We show the space usage per group of our methods on a log-scale in Figure 2(d). Undecayed methods store 4 byte integers, while forward decay stores 8 byte floating point values. The exponential histogram methods must track a large amount of information, of the order of kilobytes. This is a major factor for our queries, since they typically generate tens of thousands of groups (in the query above, there is one group for every distinct TCP destination seen in a minute on a busy link).

Random Sampling. Our experiments on drawing random samples are shown in Figure 3. The sampling techniques are all implemented as UDAFs in C code, which are then called by GSQL queries as

```
select tb, PRISAMP(srcIP, exp(time % 60))
from TCP
group by time/60 as tb
```

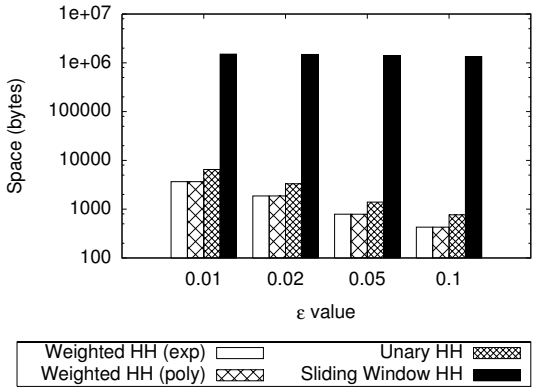
In this query, a sample is drawn every minute, with the landmark set to zero seconds within that minute. `PRISAMP` references the priority sampling UDAF (in this case), which is passed the (exponential) weight of the timestamp of the tuple.

We compare computing a fixed-size reservoir sample without decay to the two algorithms designed to draw a sample under exponential decay. Figure 3(a) shows the CPU usage as the stream data rate was varied from 100,000 to 400,000 packets per second. This plot shows only the cost of sample maintenance, and not the cost of the running selection operator which filters out TCP traffic, since this cost is the same for all algorithms. All three algorithms scale well and experience less than 10% increase in CPU load as the data rates increases from 100,000 to 400,000. The CPU load is comparable for all algorithms, meaning that we can achieve the more flexible result of the forward based decay (arbitrary timestamp values, and arbitrary arrival order) at virtually no cost over the previous solution. Moreover, Figure 3(b) shows that the cost of the three sampling methods all appear independent of the sample size. (Note that the space used by the methods is essentially that of size of the sample, plus some small additional values such as

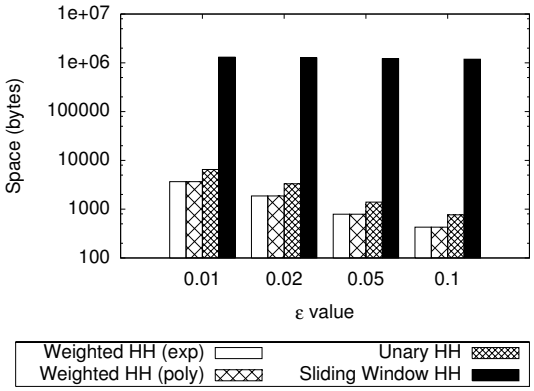


(a) CPU load as ϵ varies (TCP data)

(b) CPU load as ϵ varies (UDP data)



(c) Space usage as ϵ varies (TCP data)



(d) Space usage as ϵ varies (UDP data)

Fig. 4. Experiments on Heavy Hitter queries under time decay

stored priority values, so we do not show any plots of space used). Note that we can obtain samples under other forward decay functions at the same cost, whereas exponential decay is the only backward class model for which efficient sampling algorithms are known.

Heavy Hitter Aggregates. Our experiments on holistic aggregate computation concentrated on finding heavy hitters. For each one minute interval, the query identifies a set of network hosts receiving the most TCP traffic. We show the dominant cost, of maintaining the summary under updates, and do not plot the small final cost of extracting the heavy hitters. We varied the stream rate from 50,000 packets/sec to 200,000 packets/sec and observed the total CPU load. For forward-decayed aggregates, we compared both exponential and quadratic decay as before.

Figure 5 shows that the overhead of the weighted version of the heavy hitters algorithm is small compared to version optimized for unweighted updates ("Unary HH"). We also see that there is little variation as a function of the decay function. As we argued in our introductory analysis, the sliding window-based implementation of backward decay is much more expensive due to the complexity of the associated algorithms. At 200,000 packets/sec, the system reached 90% CPU utilization (nearing instability), and further increases in the data rate caused tuple dropping. Although it allows arbitrary decay functions to be specified at query times, this form of backward decay is simply not practical to run in a streaming system.

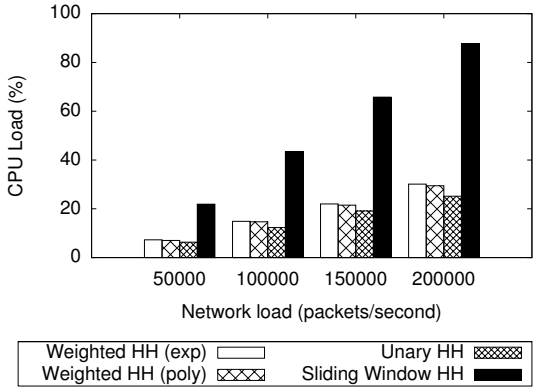


Fig. 5. HH performance as stream rate varies

This is further highlighted in Figures 4(a) and 4(c), which

show CPU and space usage (log scale) respectively as ϵ varies. The stream data rate here was set to 200,000 packets/sec using flow sampling on the network card. At $\epsilon = 0.01$, the backward decayed algorithms approach 100% CPU utilization and further increases in data rate cause tuple drops. The CPU usage of the weighted algorithms implementing forward decay is fairly robust to the value of ϵ, and the space depends on $1/\epsilon$ (the space is still of the order of kilobytes, but one typically expects such aggregate queries to be run over somewhat fewer groups than sum or count queries). Note that the space of the backward decayed approach does not vary with ϵ: this is because it does not have much pruning power over the number of tuples presented, and so it is effectively storing a large fraction of the total input. This is also unsustainable in a high-throughput streaming system.

Lastly, Figures 4(b) and 4(d) show the same experiments performed over UDP data. Here, we took the same query over only the UDP traffic (specified by adding an additional selection to the query). The stream data rate was set to 170,000 packets/sec, while the rest of the experimental settings were the same as in previous experiments. We see that the behavior of the algorithm is virtually unchanged despite the different characteristics of UDP data. The space required by Sliding Window approach is slightly lower, but still orders of magnitude higher than that for forward decay (about a megabyte compared to 1KB–6KB, depending on ϵ).

IX. Concluding Remarks

We have proposed a new class of time decay for streaming systems, based on a forward view of the decay. Just as with previous definitions, it can be motivated by metaphors with the physical world, such as radioactive decay, and perspective shrinking. It is effective to implement in streaming systems, and has a low overhead compared to processing undecayed queries, making it much more attractive than prior algorithms.

One feature of the decay is that it fits easily into distributed systems seeing different parts of an input that is to be combined. It will be interesting to study how to integrate this model of time decay into not just distributed streaming systems, such as Borealis [1], but also the new generation of popular distributed processing systems such as MapReduce [18], Hadoop [23] and Sawzall [32].

References

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Rasin, N. Tatbul, Y. Xing, and S. Zdonik. Distributed operation in the borealis stream processing engine. In *SIGMOD*, 2005.

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

[3] N. Alon, N. Duffield, C. Lund, and M. Thorup. Estimating sums of arbitrary selections with few probes. In *PODS*, 2005.

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

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

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

[7] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *VLDB*, 2002.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *SIGMOD*, 2003.

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

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

[11] G. Cormode, F. Korn, and S. Tirthapura. Exponentially decayed aggregates on data streams. In *ICDE*, 2008.

[12] G. Cormode, F. Korn, and S. Tirthapura. Time decaying aggregates in out-of-order streams. In *PODS*, 2008.

[13] G. Cormode and S. Muthukrishnan. Estimating dominance norms of multiple data streams. In *ESA*, 2003.

[14] G. Cormode, S. Tirthapura, and B. Xu. Time-decaying sketches for sensor data aggregation. In *PODC*, 2007.

[15] C. Cranor, L. Gao, T. Johnson, and O. Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *SIGMOD*, 2002.

[16] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, 2003.

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

[18] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[19] P. S. Efraimidis and P. G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters (IPL)*, 97:181–185, 2006.

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

[21] L. Golab. *Sliding Window Query Processing over Data Streams*. PhD thesis, University of Waterloo, 2006.

[22] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *FOCS*, 2000.

[23] Hadoop. `http://hadoop.apache.org/`.

[24] P. Indyk. Better algorithms for high-dimensional proximity problems via asymmetric embeddings. In *SODA*, 2003.

[25] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in gigascope. In *VLDB*, 2005.

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

[27] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[28] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *ICDE*, pages 767–778, 2005.

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

[30] F. Olken. *Random Sampling from Databases*. PhD thesis, Berkeley, 1997.

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

[32] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Dynamic Grids and Worldwide Computing*, 13(4):277–298, 2005.

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

[34] Streambase. `http://www.streambase.com/`.

[35] Stanford stream data manager. `http://www-db.stanford.edu/stream/sqr`.

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

[37] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, Mar. 1985.

[38] S. Zdonik, M. Stonebraker, M. Cherniack, and U. Cetintemel. The Aurora and Medusa projects. *Bulletin of the Technical Committee on Data Engineering*, pages 3–10, Mar. 2003.