# Exponentially Decayed Aggregates on Data Streams

Graham Cormode[†], Flip Korn[†], Srikanta Tirthapura[*]

[†]*AT&T Labs-Research*
{graham,flip}@research.att.com

[*] *Iowa State University*
snt@iastate.edu

*Abstract*— **In a massive stream of sequential events such as stock feeds, sensor readings, or IP traffic measurements, tuples pertaining to recent events are typically more important than older ones. It is important to compute various aggregates over such streams after applying a decay function which assigns weights to tuples based on their age. We focus on the computation of exponentially decayed aggregates in the form of quantiles and heavy hitters. Our techniques are based on extending existing data stream summaries, such as the q-digest [1] and the "space-saving" algorithm [2]. Our experiments confirm that our methods can be applied in practice, and have similar space and time costs to the non-decayed aggregate computation.**

## I. Introduction

The rapid growth in data volumes from applications such as networking, scientific experiments and automated processes continues to surpass our ability to store and process using traditional means. Consequently, a new generation of systems and algorithms has been developed, under the banner of "data streaming". Here, we must be able to answer potentially complex queries continuously in real time, as the stream is observed in whatever order it arrives. In contrast with a stored database, events in a data stream that have occurred recently are usually more significant than those in the distant past. This is typically handled through *decay functions* that assign greater weight to more recent elements in the computation of aggregates. Of particular interest is the notion of *exponential decay*, where the weight of an item which occurred $a$ time units ago is $\exp(-\lambda a)$, for a decay parameter $\lambda$.

Exponential decay is a popular model due in part to the relative simplicity with which simple counters can incorporate exponential decay (this result is virtually folklore). As a consequence, for methods based on counts that are linear functions of the input, such as randomized sketches, the ability to apply exponential decay and out-of-order arrivals follows almost immediately. For other summaries, this is not so immediate: Manjhi *et al.* [3] carefully prove variations of known frequent items algorithms to track heavy hitters with exponential decay in space $O(\frac{1}{\epsilon})$. Aggarwal [4] shows how to draw a sample with approximately exponential decay on sequence numbers. Other decay functions have been studied, for more details see [5], [6].

In this paper, we study how exponential decay can be applied to complex streaming aggregates, in particular quantiles and heavy hitters. Our algorithms extend previously known algorithms for these problems without any time decay, and as a consequence run with at least the same time cost and space bounds. This yields the first known deterministic algorithms for quantiles under exponential decay; it also yields algorithms for heavy hitters under exponential decay which are simpler and more flexible than prior work [3], since they tolerate arrivals in arbtiirary orders. As such, they are quite practical for use in a live data streaming system handling many hundreds of thousands of transactions per second.

## II. Preliminaries

*Definition 1:* A data stream is an (unbounded) sequence of tuples $e_i = \langle x_i, w_i, t_i \rangle$, where $x_i$ is the identifier of the item (the key), $w_i$ is a non-negative initial weight associated with the item, and $t_i$ the timestamp.

For example, a stream of IP network packets may be abstracted as a stream where $x_i$ is the destination address, $w_i$ is the size of the packet in bytes, and $t_i$ the time at which it was sent. The "current time" is denoted by the variable $t$. It is possible for many items in the stream to have the same timestamp. The weight of an item at time $t$ is based on an exponential decay function:

*Definition 2:* Given an input stream $S = \{\langle x_i, w_i, t_i \rangle\}$, the decayed weight of each item at time $t$ is $w_i \exp(-\lambda(t - t_i))$ for a parameter $\lambda > 0$. The *decayed count* of the stream at $t$ is $D(t) = \sum_i w_i \exp(-\lambda(t-t_i))$ (or just $D$ when $t$ is implicit).

The definitions of time-decayed aggregates introduced below are implicit in some prior work, but have not previously been stated explicitly. In most cases, the definitions of time-decayed aggregates are natural and straightforward extensions of their undecayed versions. Since exact computation of these aggregates requires space linear in the input size even without decay, we consider the following approximation problems:

*Definition 3:* For $0 < \epsilon < \phi \leq 1$, the *$\epsilon$-approximate exponentially decayed $\phi$-quantiles problem* is to find $q$ so that
$$(\phi - \epsilon)D \leq \sum_{i, x_i < q} w_i \exp(-\lambda(t - t_i)) \leq (\phi + \epsilon)D.$$

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

Note that timestamp $t_i$ is completely decoupled from time $t$ when the tuple is observed. So it is possible that $i < j$, so that $e_i = \langle x_i, w_i, t_i \rangle$ is received earlier than $e_j = \langle x_j, w_j, t_j \rangle$, but $t_i > t_j$ so $e_i$ is more recent than $e_j$. The above aggregates are thus well defined on such out-of-order arrivals.

## III. Exponentially Decayed Quantiles

We describe our approach for computing quantiles on timestamp ordered data under exponential decay, which is the first deterministic algorithm for this problem. Given a parameter $0 < \epsilon < 1$, the q-digest [1] summarizes the frequency distribution $f_i$ of a multiset defined by a stream of $N$ items drawn from the domain $[0 \ldots W-1]$. The q-digest can be used to estimate the *rank* $r(q)$ of an item $q$, which is defined as the number of items dominated by $q$, i.e., $\mathrm{r}(q) = \sum_{i<q} f_i$. The data structure maintains an appropriately defined set of *dyadic ranges* of the form $[i2^j \ldots (i+1)2^j - 1]$ and their associated counts. It is easy to see that an arbitrary range of integers $[a \ldots b]$ can be uniquely partitioned into at most $2\log(b-a)$ dyadic ranges, with at most 2 dyadic ranges of each length. The q-digest has the following properties:

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

Given query point $q \in [0 \ldots W-1]$, we can compute an estimate of the rank of $q$, denoted by $\hat{r}(q)$, as the sum of the counts of all ranges to the left of $q$, i.e. $\hat{r}(q) = \sum_{(r=[l,h],c(r)),h<q} c(r)$. The following accuracy guarantee can be shown for the estimate of the rank: $\hat{r}(q) \leq \mathrm{r}(q) \leq \hat{r}(q) + \epsilon N$. Similarly, given a query point $q$ one can estimate $f_q$, the frequency of item $q$ as $\hat{f}_q = \hat{r}(q+1) - \hat{r}(q)$, with the following accuracy guarantee: $\hat{f}_q - \epsilon N \leq f_q \leq \hat{f}_q + \epsilon N$. The q-digest can be maintained in space $O(\frac{\log W}{\epsilon})$ [1], [7]. Updates to a q-digest can be performed in (amortized) time $O(\log\log W)$, by binary searching the $O(\log W)$ dyadic ranges containing the new item to find the appropriate place to record its count; and queries take $O(\frac{\log W}{\epsilon})$. Now observe that: (1) The q-digest can be modified to accept updates with arbitrary (i.e. fractional) non-negative weights; and (2) multiplying all counts in the data structure by a constant $\gamma$ gives an accurate summary of the input scaled by $\gamma$. It is easy to check that the properties of the data structure still hold after these transformations, e.g. that the sum of the counts is $D$, the sum of the (possibly scaled) input weights; no count for a range exceeds $\frac{\epsilon D}{\log U}$; etc.

Thus given an item arrival of $\langle x_i, t_i \rangle$ at time $t$, we can create a summary of the exponentially decayed data. Let $t'$ be the last time the data structure was updated; we multiply every count in the data structure by the scalar $\exp(-\lambda(t-t'))$ so that it reflects the current decayed weights of all items, and then update the q-digest with the item $x_i$ with weight $\exp(-\lambda(t-t_i))$. Note that this may be time consuming, since it affects every entry in the data structure. We can be more "lazy" by tracking $D$, the current decayed count, exactly, and keeping a timestamp $t_r$ on each counter $c(r)$ denoting the last time it was touched. Whenever we require the current value of range $r$, we can multiply it by $\exp(-\lambda(t-t_r))$, and update $t_r$ to $t$. This ensures that the asymptotic space and time costs of maintaining an exponentially decayed q-digest are as before.

**Algorithm IV.1:** HeavyHitterUpdate$(x_i, w_i, t_i, \lambda)$

```
Input: item $x_i$, timestamp $t_i$, weight $w_i$, decay factor $\lambda$
Output: Current estimate of item weight
if $\exists j.\, \mathrm{item}[j] = x_i$;
  then $j \leftarrow \mathrm{item}^{-1}(x_i)$
  else $j \leftarrow \arg\min_k(\mathrm{count}[k])$;
$\mathrm{item}[j] \leftarrow x_i$;
$\mathrm{count}[j] \leftarrow \mathrm{count}[j] + w_i \exp(\lambda t_i)$
return $(\mathrm{count}[j] * \exp(-\lambda t_i))$
```

Fig. 1. Pseudocode for Heavy Hitters with exponential decay

To see the correctness of this approach, let $S(r)$ denote the subset of input items which the algorithm is representing by the range $r$: when the algorithm processes a new update $\langle x_i, t_i \rangle$ and updates a range $r$, we (notionally) set $S(r) = S(r) \cup i$; when the algorithm merges a range $r'$ together into range $r$ by adding the count of (the child range) $r'$ into the count of $r$ (the parent), we set $S(r) = S(r) \cup S(r')$, and $S(r') = \emptyset$ (since $r'$ has given up its contents). Our algorithm maintains $c(r) = \sum_{i \in S(r)} w_i \exp(-\lambda(t-t_i))$; it is easy to check that every operation which modifies the counts (adding a new item, merging two range counts, applying the decay functions) maintains this invariant. In line with the original q-digest algorithm, every item summarized in $S(r)$ is a member of the range $r$, i.e. $i \in S(r) \Rightarrow x_i \in r$, and at any time each tuple $i$ from the input is represented in exactly one range $r$.

To estimate the decayed rank of $x$ at time $t$, $\mathrm{r}_\lambda(x,t) = \sum_{i, x_i \leq x} w_i \exp(\lambda(t-t_i))$, we compute

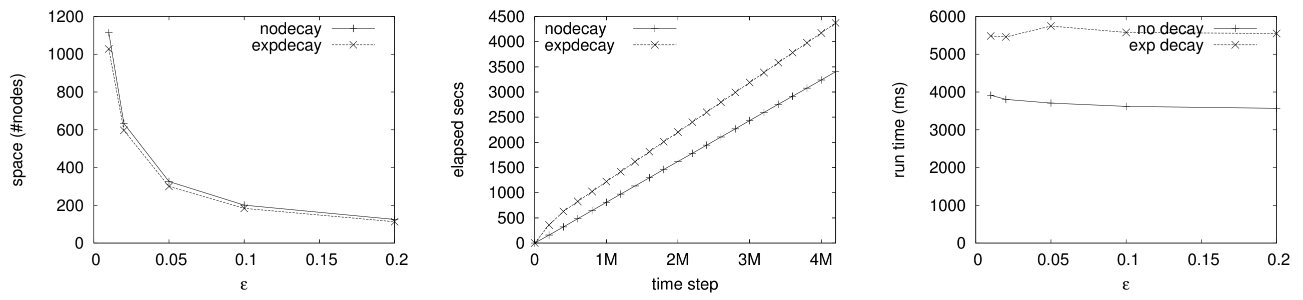
$$\hat{r}_\lambda(x,t) = \sum_{r=[l \ldots h], h \leq x} c(r).$$

By the above analysis of $c(r)$, we correctly include all items that are surely less than $x$, and omit all items that are surely greater than $x$. The uncertainty depends only on the ranges containing $x$, and the sum of these ranges is at most $\epsilon \sum_r c(r) = \epsilon D$. This allows to quickly find a $\phi$-quantile with the desired error bounds by binary searching for $x$ whose approximate rank is $\phi D$. In summary,

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

## IV. Exponentially Decayed Heavy Hitters

Prior work by Manjhi *et al.* [3] computed Heavy Hitters on timestamp ordered data under exponential decay by modifying algorithms for the problem without decay. We take a similar tack, but our approach means that we can also easily accommodate out-of-order arrivals, which is not the case in [3]. A first observation is that we can use the same (exponentially decayed) q-digest data structure to also answer heavy hitters queries, since the data structure guarantees error at most $\epsilon D$ in the count of any single item; it is straightforward to scan the data structure to find and estimate all possible heavy hitters in time linear in the data structure's size. Thus Theorem 1 also applies to heavy hitters. However, we can reduce the required



(a) exp ($\lambda = 0.1$) and no-decay: space vs $\epsilon$ (b) performance vs timestamp (World Cup) (c) performance vs $\epsilon$ (flow)

Fig. 2. Experimental results on real data for exponentially time-decayed aggregates

space, and extend to the case when the input is drawn from an arbitrary domain, and arrives in arbitrary order.

Our algorithm, a modified version of the "Space-saving" algorithm [2] tracks a set of $O(\frac{1}{\epsilon})$ pairs of item names and counters, with the counters initialized to zero. For each item $x_i$ in the stream, we see if there is currently an (item, counter) pair for that item. If so, we update the quantity of $w_i \exp(\lambda t_i)$, and add this to the counter associated with $x_i$. Otherwise, we add the same quantity, $w_i \exp(\lambda t_i)$, to the *smallest* counter (breaking ties arbitrarily), and set the item associated with the counter to $x_i$. Pseudo-code is in Figure 1. To find the heavy hitters, visit each item stored in the data structure, $\mathrm{item}[i]$, estimate its decayed weight at time $t$ as $\exp(-\lambda t)\,\mathrm{count}[i]$, and output $\mathrm{item}[i]$ if this is above $\phi D$.

*Theorem 2:* The algorithm finds $\epsilon$-approximate exponentially decayed heavy hitters in space $O(\frac{1}{\epsilon})$, with update time $O(\log \frac{1}{\epsilon})$. Queries take time $O(\frac{1}{\epsilon})$.

*Proof:* The space bound follows from the definition of the algorithm and the time bound follows if we use a standard heap data structure to track the smallest count in the data structure. It remains to prove correctness. The following invariant holds by induction over updates: $\sum_j \mathrm{count}[j] = \sum_i w_i \exp(\lambda t_i) = D\exp(\lambda t)$. Also, since there are $\frac{1}{\epsilon}$ counters, the smallest count, $min = \min_j(\mathrm{count}[j])$, is at most $\epsilon D \exp(\lambda t)$. The true decayed count of any item $x_i$ which is not recorded in the data structure is at most $min \exp(-\lambda t)$, by induction over the sequence of updates (it is true intially, and remains true over each operation). Thus, when an uncounted item is stored in the data structure, we associate it with the current value of $min$, which at is always an overestimate of the true decayed count (scaled by $\exp(\lambda t)$). Hence, every count at query time is an overestimate. Suppose we do not store some item $x$ whose true decayed count is above $\epsilon D$. Then we must have overwritten $x$ with another item $x_i$ when the minimum count was $min_i$. But since our estimate of the count of $x$ at any instant is guaranteed to be an overestimate, this gives a contradiction, since the true decayed count of $x$ is at least $\epsilon D \geq min \geq min_i$; consequently, we would not have overwritten $x$. Therefore, we can conclude that the data structure retains information about all items with decayed count at least $\epsilon D$. The estimated counts are overestimates by at most $\epsilon D$, so we can accurately answer heavy hitter queries from the stored information. ■

## V. Experiments

We implemented our method from Section III (based on q-digests) in C and measured the space usage (in terms of number of nodes stored in the data structure) and processing time. We show results on two different network data streams: 5 million records of IP flow data aggregated at an ISP router using Cisco NetFlow, projected onto `(begin_time, num_octets)`; and 5 million records of Web log data collected during the 1998 Football World Cup (`http://ita.ee.lbl.gov/.`), projected onto `(time, num_bytes)`. Experiments were run on a 2.8GHz Pentium Linux machine with 2 GB main memory.

Figure 2(a) graphs the space usage of exponential decay compared to no decay (regular q-digests) for different values of $\epsilon$ on flow data. It shows that in practice there is very little space overhead for exponential decay. The results on World Cup data (not shown) were almost identical. Figure 2(b) compares the time (in seconds) taken to update the data structure for exponential and no-decay at increasing timestamps using World Cup data (flow data was similar). Figure 2(c) shows how these times vary with $\epsilon$, on a log scale. Exponential decay can handle a throughput of around 1 million updates per second. It is highly effective to implement, since the overhead compared to no decay is small.

## References

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

[2] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International Conference on Database Theory*, 2005.

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

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

[5] E. Cohen and M. Strauss, "Maintaining time-decaying stream aggregates," in *ACM Principles of Database Systems*, 2003.

[6] G. Cormode, F. Korn, and S. Tirthapura, "Time decaying aggregates in out-of-order streams," Center for Discrete Mathematics and Computer Science (DIMACS), Tech. Rep. 2007-10, 2007.

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