2009

A study of time-decayed aggregates computation on data streams

Bojian Xu
Iowa State University

A study of time-decayed aggregates computation on data streams

by

Bojian Xu

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Srikanta Tirthapura, Major Professor
Pavan Aduri
Soma Chaudhuri
Daji Qiao
Zhao Zhang

Iowa State University
Ames, Iowa
2009

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

First and most, I want to thank my advisor Srikanta Tirthapura for leading me into the area and giving me so much freedom in doing research. His standard for quality work and enormous patience for new discoveries make my doctoral study an enjoyable experience. His generous support also enables me to be focused on my study. Graham Cormode was an excellent mentor when I visited AT&T as an intern in Summer 2008. I want to thank him for his research discussion and collaboration since then. I want to thank Costas Busch, Vladislav Shkapenyuk and Divesh Srivastava for their research contributions in this work. I also want to thank Chris Chu for being my advisor in the Preparing Future Faculty program. His being a good role model has often inspired me to be a better student and researcher. I also owe him for squeezing time from his busy schedule to review my job application documents. I would also like to thank my committee members, Dr. Pavan Aduri, Dr. Soma Chaudhuri, Dr. Daji Qiao and Dr. Zhao Zhang, for their advice in this work.

I want to thank my officemates, Tycho Anderson, Bibudh Lahiri, Puvi Pandian, Zhenhui Shen and Jason Stanek, for the research discussions and the fun in the river fishing together. I want to thank my roommates, Guanghai Ji, Rong Jiao, Scott Polifka, Song Sun and Kewei Tu, for their friendship. I also want to thank the brothers and sisters in Christ in TCCF and CEFCA, for their encouragement during my stay in US.

I want to appreciate my parents Qingyue Xu and Suqing Hong for their unconditional love. I will never have a chance to be who I am without their endless giving. I also want to thank my sisters Sili and Sijian and their families, who I have been missing so much all the time, for their being part of my loving family in China. Finally, I want to thank my wife, Weili, for her love and silent support, and more sweetness in the future.

Finally but not least, I want to thank the Almighty Lord, who I am still pursuing, for giving me the purpose and strength to keep growing. To him be all the glory.

ABSTRACT

Many real world data naturally arrive as rapid paced and virtually unbounded *streams*. Examples of such streams include network traffic at a router, events observed by a sensor network, accesses to a web server and transactional updates to a large database. Such streaming data need to be monitored online to collect traffic statistics, detect trends and anomalies, tune system performance and help make business decisions. However, because of the large size and rapid pace of the data, as well as the online processing requirement, conventional data processing methods, such as storing the data in a database and issuing offline SQL queries thereafter, are not feasible. Data stream processing is a new diagram of massive data set processing and creates new challenges in the algorithm design and implementation.

In this thesis, we consider *time-decayed* data aggregation for data streams, where the importance or contribution of each data element decays over time, since recent data are usually considered of more importance in applications, and therefore are given heavier weights. We design small space data structures and algorithms for maintaining fundamental aggregates of the streams if it is possible and otherwise show large space lower bounds.We consider the data aggregation over a robust data stream model called *asynchronous data stream*, motivated by the streaming data transmitted in distributed systems, including computer networks, where the asynchrony in the data transmission is inevitable. In asynchronous data stream, the arrival order of the data elements at the receiver side is not necessarily the same as the order in which the data elements were generated. Asynchronous data stream is a robuster and generalized model of the previous synchronous data stream model.

In summary, this thesis presents the following results:

- We formalize the model of asynchronous data stream and the notion of timestamp sliding window. We propose the first small space *sketch* for summarizing the data elements over timestamp sliding windows of multiple geographically distributed asynchronous data streams. The sketch can return accuracy guaranteed estimates for basic aggregates, such as: Sum, Median and Quan-

tiles.

- We design the first small space sketch for general purpose network streaming data aggregation. The sketch has the following properties that make it useful in communication-efficient aggregation in distributed streaming scenarios: (1) The sketch can handle multiple geographically distributed asynchronous data streams. (2) The sketch is duplicate-insensitive, i.e. reinsertions of the same data will not affect the sketch, and hence the estimates of aggregates. (3) The sketch is also time-decaying, so that the weight of each data element summarized in the sketch decreases over time. (4) The sketch returns accuracy guaranteed estimates for a variety of core aggregates, including the sum, median, quantiles, frequent elements and selectivity.

- We conduct a comprehensive study on the time-decayed *correlated* data aggregation over asynchronous data streams. For each class of time decay function, we either propose space efficient algorithms or show large space lower bounds. We not only closes the open problem of correlated data aggregation under sliding windows decay, but also presents negative results for the case of exponential decay, which however is highly used in the non-correlated scenarios.

- We propose the *forward decay* model to simplify the time-decayed data stream aggregation and sampling. Forward decay captures a variety of usual decay functions (or called *backward decay*), such as exponential decay. We design efficient algorithms for data aggregation and sampling under the forward decay model, and show that they are easy to implement scalably.

CHAPTER 1. Introduction

Many real world data naturally arrive as *streams*. Examples include network traffic at a router, events observed by the sensor motes of a wireless sensor network, webpage requests to a web server and transactional updates to a large database. Contributed by the advancement of modern computer and Internet technologies, such streaming data has became highly paced and massive, compared to the ability of computing, storing and transmitting that the data processor can provide. For example, an OC48 link has its standard transmission rate at 2.5 Gbits per second, much more than the storage capacity of a normal computer, and the flowing data are even too quick to take a scan of it; In the recently emerging wireless sensor network applications, events observed by the battery powered tiny sensor mote can quickly overwhelm the mote's memory.

However, these streaming data need to be monitored to collect traffic statistics, detect trends and anomalies, tune system performance and even help make business decisions. In some applications, realtime queries and answers are even demanded. For example, a search engine may want to continuously maintain a list of hot searching keywords mined from the massive streams of searching queries that the search engine has received from the users. However, because of the large size and rapid pace of the data as well as the demands for realtime queries and answers, conventional data processing methods, such as storing the data in a database and issuing offline SQL queries, are not feasible.

The goal in data stream processing research is to answer questions like: Given such amount of time and space, and a data steam or multiple streams arriving at such a pace, what query about the streams can we answer ? If not, can we give an approximate answer for the query ? How accurate the approximate answer can we guarantee ? Or can we show that the problem is inherently unsolvable in a streaming fashion using such limited space and time budget ? The typical challenges in the algorithms design and implementation for the data stream processing are as follows.

1. The algorithms must use small workspace. Because of the limited memory space and the virtually

unbounded stream size, the space cost of the algorithm usually must be poly-logarithmic in the size of the streams and sometimes even independent from the stream size. A small workspace also help in processing stream elements more quickly by reducing the time cost in scanning and searching through the data structures.

2. The algorithm must process each stream element quickly in order to keep up with the pace of the stream.

3. Constrained by the small space budget, one-pass processing of the stream is required, since we are incapable to store the majority of the stream and therefore have no chance to visit old data elements. One-pass processing is also needed to support continuous queries.

4. Queries of interest can be submitted at anytime during the stream processing. Answers should be returned immediately regarding the data that have been received so far. Answers for the queries can comprise a new stream and can be the input of another stream processor.

5. In many scenarios, the sacrifice of fast processing and small workspace is to lose the accuracy in the answers for queries. Although approximate answers are acceptable in many applications, error in these approximations need to be well bounded.

1.1 Asynchronous Data Streams

In this thesis, we particularly focus on *asynchronous data streams* (Definition 1.3.1), motivated by the data streams transmitted in distributed systems including networks. In distributed stream processing, it is necessary to deal with the inherent asynchrony in the network through which data is being transmitted. Nodes often have to process composite data streams that consist of interleaved data from multiple data sources. One consequence of the network asynchrony is that in such composite data streams, the arrival order of the stream elements is not necessarily the same as the order in which the elements were generated. We call such a data stream as an asynchronous data stream.

Asynchronous data streams are inevitable anytime two streams, say A and B, fuse with each other and the data processing has to be done on the stream formed by the interleaving of A and B. Even if individual streams A or B are not inherently asynchronous, when the streams are fused, the stream

could become asynchronous. For example, if the network delay in receiving stream *B* is greater than the delay in receiving elements in stream *A*, then the the stream processor may consistently observe elements with earlier timestamps from *B* after elements with more recent timestamps from *A*.

In asynchronous data streams the order of "recency" of the data may not be preserved. The notion of recency can be captured with the help of a timestamp associated with the stream element. The greater the timestamp of an element is, the more recent the element is. Asynchronous stream is a more natural model for data streams transmitted in distributed systems than the synchronous stream model, and it is therefore robuster for distributed data stream monitoring.

1.2 Distributed Data Streams Processing Diagram

In applications involving distributed data sources, such as content distribution, intranet monitoring, and sensor data processing, no single node observes all data, yet aggregates should be computed over the union of the data observed at all the nodes. Therefore, it is necessary to answer aggregate queries for the union of all the streams distributively. A naive approach to solve such problems is to send all streams to a single aggregator. However, this approach is too costly, since there is a communication and energy cost for every data item in every stream. Thus, the data streams have to be combined in a more efficient way in order to minimize the use of network resources. This is critical especially in sensor networks where nodes are typically battery operated devices.

In our approach, we place an aggregator for each stream. Each aggregator maintains a small space *sketch* summarizing its local stream. All the sketches for local streams can be combined distributively to create the sketch for the union of streams (Figure 2.1). The sketch for the union can be used to answer queries regarding the union of streams. The sketches of local streams will be combined in a compact and lossless way, i.e., the space complexity and accuracy guarantee of the sketch for the union is the same as those for local streams. Also, the sketch for the union can be constructed on demand, whenever new queries are issued. Unlike previous work (42; 41; 57) that considered the synchronous model on distributed streams, this thesis considers aggregate computation over distributed streams under asynchronous arrival.

1.3 Data Stream Model

We model a stream as $R = \langle e_1, e_2, \ldots, e_n \rangle$, where e_i is received earlier than e_j for any pair of i and j, $i < j$. Note that e_n is the most recently received element and n can be infinitely large. Each stream element 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
- w_i is a weight associated with the value
- t_i is the integer timestamp, tagged at the time e_i was created.
- id_i is a unique 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 more examples in Chapter 3.

We consider asynchronous data stream. In other words, it is possible that in stream R an element tagged with a larger (and thus newer) timestamp is received earlier than an element of a smaller (and thus older) timestamp. More formally, we define an asynchronous data stream as follows.

Definition 1.3.1 (Asynchronous Data Stream)**.** *Stream R is an asynchronous data stream if for any pair of i and j, $1 \leq i < j \leq n$, it is possible that $t_i > t_j$.*

Our stream model also allows the possibility that the same observation appears multiple times in the stream, with the same id, value, weight and timestamp preserved across multiple appearances. Such occurrences exist in real network environment due to the multi-path routing to increase the chance of data delivery, yet only one copy of such repeated occurrences should 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, weights and/or timestamps — in such a case, they will be considered separately in computing the aggregates.

1.4 Time Decay

In most evolving settings, recent data is more reliable or more important than older data. We should therefore weigh newer stream elements (with larger timestamps) more heavily than older ones. This can be formalized in a variety of ways: we may only consider data elements that fall within a *sliding window* of recent time (say, the last hour), and ignore (assign zero weight to) any that are older (35); or, more generally, use an arbitrary function that assigns a weight to each data element as a function of its initial weight and age (21).

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 1.4.1. *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 $f(w,c-t)$. In Chapter 5, we will also call such decay function as backward decay, since the value of it depends on the element's age which is computed by looking from the current time backward to the element's timestamp.

1.4.1 Decomposable Decay Functions

Decomposable decay is a class of decay functions and is popularly used in applications.

Definition 1.4.2. *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 $g(x_1) \geq g(x_2)$. In the rest of this thesis, we will also call $g(x)$ decay function if the context is clear. The following are example decomposable decay functions.

No decay. The trivial function $g(x) \equiv 1$ 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 $g(x) = 1$ for $x \leq W$ and $g(x) = 0$ for $x > W$ captures the common sliding window semantics—only items whose age is less than or equal to W are considered.

Polynomial decay. Given a constant $a > 0$, the polynomial decay function is defined as $g(x) = (x+1)^{-a}$.

Exponential decay. Given a constant $\alpha > 0$, the exponential decay function is defined as $g(x) = 2^{-\alpha x}$. Exponential decay with a different base can also be written in this form, since $a^{-\lambda x} = 2^{-\lambda \log_2(a) x}$.

Super-exponential decay. A decay function $g(x)$ is super-exponential, if there exist constants $\sigma > 1$ and $c \geq 0$, such that for every $x \geq c$, $f(x)/f(x+1) \geq \sigma$. Examples of such decay functions include: (1) *polyexponential decay* (21): $g(x) = (x+1)^k 2^{-\alpha x}/k!$ where $k > 0$, and $\alpha > 0$ are constants. (2) $g(x) = 2^{-\alpha x^\beta}$, where $\alpha > 0$ and $\beta > 1$.

Converging decay. A decay function $g(x)$ is a converging decay function if $g(x+1)/g(x)$ is non-decreasing with x. Intuitively, the relative weights of elements with different timestamps under a converging decay function get closer to each other as time goes by. As pointed out by Cohen and Strauss (21), this is an intuitive property of a time-decay function in several applications. Many popular decay functions, such as exponential decay and polynomial decay, are converging decay. Converging decay also includes the no decay case: $g(x) \equiv 1$.

Finite decay. A decay function is defined to be a finite decay function with age limit N, if there exists $N \geq 0$ such that for $x > N$, $g(x) = 0$, and for $x \leq N$, $g(x) > 0$. Examples of finite decay include (1) sliding window decay, where the age limit N is the window size. (2) Chordal decay with an age limit $N-1$ (21): $g(x) = 1 - x/N$ if $0 \leq x \leq N$ and $g(x) = 0$ otherwise. Obviously, no finite decay function is a converging decay function, since $g(N+1)/g(N) = 0$ while $g(N)/g(N-1) > 0$.

1.5 Thesis Contributions

In this thesis, we focus on time-decayed asynchronous data stream processing. We design time and space efficient algorithms for data aggregations in the setting of distributed data streams. We also show large space lower bounds for problems that are inherently hard. The following are the main contributions of this thesis.

- We formalize the model of asynchronous data stream and the notion of timestamp sliding window. We propose the first small space *sketch* for summarizing the data elements over timestamp sliding windows of multiple geographically distributed asynchronous data streams. The sketch can return accuracy guaranteed estimates for basic aggregates, such as: Sum, Median and Quantiles. (Chapter 2)

- We propose the first small space sketch for general purpose network data aggregation. The sketch has the following properties that make it useful in communication-efficient aggregation in distributed streaming scenarios: (1) The sketch can handle asynchronous data streams. (2) The sketch is duplicate-insensitive, i.e. reinsertions of the same data will not affect the sketch, and hence the estimates of aggregates. (3) The sketch is also time-decaying, so that the weight of each data element summarized in the sketch decreases over time according to any arbitrary user-specified decay function. (4) The sketch can give provably approximate guarantees for a variety of core aggregates of data, including the sum, median, quantiles, frequent elements and selectivity. (5) The size of the sketch and the time taken to update it are both polylogarithmic in the size of the relevant data. (6) Multiple sketches computed over distributed data streams can be combined without loss of accuracy. (Chapter 3)

- We conduct a comprehensive study on the time-decayed *correlated* data aggregation over asynchronous data streams. For each class of time decay function, we either propose space efficient algorithms or show large space lower bounds. We not only closes the open problem of correlated data aggregation under sliding windows decay, but also presents negative results for the case under exponential decay, which however is highly used in the non-correlated scenarios. (Chapter 4)

- We propose the *forward decay* model to simplify the time-decayed data stream aggregation and sampling. Forward decay captures a variety of usual decay functions (or called *backward decay*), such as exponential decay and polynomial decay. We design efficient algorithms for data aggregation and sampling under the forward decay model, and show that they are easy to implement scalably. (Chapter 5)

1.6 Roadmap

In Chapter 2, we consider asynchronous data stream processing over sliding windows and design small space sketches for data aggregation. This sketch is further extended in Chapter 3 for more general purpose network streaming data aggregation. We then extend the techniques in processing asynchronous data stream for the correlated data stream aggregation in Chapter 4. In Chapter 5, we present forward decay, a new time decay model to simplify the time-decayed data stream aggregation and sampling. We conclude this thesis with several open problems in Chapter 6.

1.7 Declarations

Publications. The work presented in this thesis has been published in the following computer science conference proceedings and journals. The majority of this thesis derives from these publications.

- The work of Chapter 2 has been published in (75) and (79).
- The work of Chapter 3 has been published in (30) and (32).
- The work of Chapter 4 has been published in (31) and (29).
- The work of Chapter 5 has been published in (28).

My Contributions. Like many thesis work, my thesis research is a collaborative work with my major professor Srikanta Tirthapura and other researchers from universities and research laboratories. Here I clarify my contributions in these collaborations.

- The proposal of the asynchronous data stream model, presented in Chapter 2, is due to Srikanta Tirthapura. He also proposed the idea of sampling approach for solving the problem. I finished the technical proofs under his guidance.
- The proposal of designing a general purpose sketch for network streaming data aggregation, presented in Chapter 3, is due to Srikanta Tirthapura and myself. The main algorithmic ideas and proofs for solving the problem as well as the experimental study are due to myself.
- The proposal of the time-decayed correlated data aggregation, presented in Chapter 4, is due to Srikanta Tirthapura and myself. Most of the algorithms and lower bound proofs are due to myself. Graham Cormode improved the space lower bound for the exponential decay based correlated sum based on my discovery of the lower bound and using my proof idea.

- The work presented in Chapter 5 was done during my visit to AT&T Shannon Laboratory as an intern in Summer 2008. The idea of forward decay is due to the colleagues at AT&T. I studied the sampling techniques under the forward decay model, as well as part of the experimental study.

CHAPTER 2. Sliding Windows Decay Based Processing

In this chapter, we study the problem of maintaining sketches for the data elements in the sliding windows over an asynchronous data stream. The sketches can give provably accurate estimates of two basic aggregates, the sum and the median, of the stream of numbers in the sliding windows. The space taken by the sketches, the time needed for querying the sketches, and the time for inserting new elements into the sketches are all polylogarithmic with respect to the maximum window size. The sketches can be easily combined in a lossless and compact way, making them useful for aggregating distributed data streams. Previous works on sketching recent elements of a data stream have all considered the more restrictive scenario of synchronous streams, where the observed order of data is the same as the time order in which the data was generated.

2.1 Introduction

Beyond the asynchronous data stream model motivated by the streaming data transmitted in distributed systems as we described in Section 1.1, in many applications, only the most recent elements in the data stream are important in computing aggregates and statistics, while the old ones are not. For example, in a stream of stock market data, a software may need to track the moving average of the price of a stock over all observations made in the last hour. In network monitoring, it is useful to monitor the volume of traffic destined to a given node during the most recent window of time. In sensor networks, only the most recent sensed data might be relevant, for example, measurements of seismic activity in the past few minutes. Motivated by such applications, there has been much work (7; 38; 42; 10; 35; 57) on designing algorithms for maintaining aggregates over a *sliding window* (Section 1.4) of the most recent elements of a data stream. So far, all work on maintaining aggregates over a sliding window has assumed synchronous streams where the arrival order of the data in a stream is the same as the time

order in which the data was generated. However, this assumption may not be realistic in distributed systems, as we have explained in Section 1.1.

The challenge with maintaining aggregates over a sliding timestamp window is that the data within the window can be very large and it may be infeasible to store the data in the workspace of the aggregator. To overcome this limitation, a fundamental technique for computing aggregates is for the aggregator to keep a small space *sketch* that contains a summary representation of all the data that has arrived within the window. Typically, the size of the sketch is much smaller than the size of the data within the window. Usually, the goal is to construct sketches whose size is polylogarithmic in the size of the data within the window. The sketch is constructed in a way that it enables the efficient computation of aggregates. Since the sketch cannot keep complete information of the streams within the small space, there is an associated *relative error* with the answer provided by the sketch, in relation to the exact value of the aggregate. The size of the sketch depends on this relative error.

Data Stream and Goal. Recall that each element e_i in stream R is a tuple (v_i, w_i, t_i, id_i) (Section 1.3). In this chapter, we consider a a projection of the stream R over the dimensions of the *value* and *timestamp*. In the projected stream, each element d_i is a tuple (v_i, t_i). The goal for the aggregator who is receiving stream R is to maintain small space sketches that can continuously return answers for queries of the following form: return an aggregate (say, the sum or the average) of all elements in the current sliding window, e.g., of those received stream elements whose timestamps are within $[c-w, c]$, where c is the clock current time at any instant (Section 1.4) and w is the size of the sliding window. When the context is clear in this chapter, we will still use R to denote the projected stream, e.g. let $R = d_1, d_2, \ldots, d_n$, and use the term "sliding timestamp window" to refer to all received items that have timestamps in the range $[c-w, c]$.

2.1.1 Contributions

First, we give algorithms for computing the sum and median of the sliding timestamp window of the asynchronous stream R that is being observed by a single aggregator. We then consider the distributed case, where we give an algorithm that combines the sketches produced by the aggregators, each of which is observing and sketching a local stream. In the discussion below, let W a bound on the maximum window size.

2.1.1.1 Sum

Our first sketching algorithm estimates the sum of all integers in stream R which are within any recent timestamp window of size $w \leq W$, i.e. $V = \sum_{\{(v,t) \in R | c-w \leq t \leq c\}} v$. The algorithm maintains a sketch using small space, that can be updated quickly when a new element arrives, and can give a provably good estimate for the sum when asked. We will use the notion of an (ε, δ)-estimator to quantify the quality of answers returned by the algorithm.

Definition 2.1.1. *For parameters $0 < \varepsilon < 1$ and $0 < \delta < 1$, an (ε, δ)-estimator for a number Y is a random variable X such that $\Pr[|X - Y| > \varepsilon Y] < \delta$. The parameter ε is called the relative error and δ is called the failure probability.*

Our algorithm for the sum has the following performance guarantees.

- For any $w \leq W$ specified by the user at the time of the query, the sketch returns an (ε, δ)-estimator of V. The value of w, the window size does not need to be known when the stream is being observed and sketched. Only W, an upper bound on w needs to be known in advance. In other words, our sketch comprises information about *every* timestamp window in the stream whose right endpoint is the current time c, and whose width is less than or equal to W.
- Space used by the sketch is $O\left(\left(1/\varepsilon^2\right) \cdot \log\left(1/\delta\right) \cdot \log V_{max} \cdot \sigma\right)$, where V_{max} is an upper bound on the value of the sum V, σ is the number of bits required to store an input element (v,t), ε is the desired relative error, and δ is the desired upper bound on the failure probability.
- The time complexity for processing an element is $O(\log\log\left(1/\delta\right) + \log\left(1/\varepsilon\right))$.
- Time taken to process a query for the sum is $O\left(\left(1/\varepsilon^2\right) \cdot \log V_{max} \cdot \log\left(1/\delta\right)\right)$

An important special case of the sum of positive integers is the problem of maintaining the number of data items within the window, and is called *basic counting* (35; 42). Our algorithm solves basic counting immediately by taking $v = 1$ for every data item.

2.1.1.2 Median

The next aggregate is the approximate median. Given $w \leq W$ specified by the user, we present an algorithm that can return an approximate median of the set $R_w = \{(v,t) \in R | c - w \leq t \leq c\}$. An

(ε, δ)-approximate median is defined as follows.

Definition 2.1.2. *For $0 < \varepsilon < 1/2$ and $0 < \delta < 1$, an (ε, δ)-approximate median of a totally ordered set S is a random variable Z such that the rank of Z in S is between $(1/2 - \varepsilon)|S|$ and $(1/2 + \varepsilon)|S|$ with probability at least $1 - \delta$. The parameter ε is called the relative error and δ is called the failure probability.*

Our algorithm has the following performance guarantees.

- For any $w \leq W$ specified by the user at the time of query, the sketch returns an (ε, δ)-approximate median of the set R_w. Similar to the sum, the sketch can answer queries about any timestamp window whose right endpoint is c and whose width is less than or equal to W.
- Space used by the sketch is $O\left(\left(1/\varepsilon^2\right)\log(1/\delta) \cdot \log N_{max} \cdot \sigma\right)$, where N_{max} is an upper bound on the number of elements in R_w, σ is the number of bits required to store an input element (v,t), ε is the desired relative error, and δ is the desired upper bound on the failure probability.
- The expected time taken to process each item is $O(\log\log(1/\delta) + \log(1/\varepsilon))$.
- Time taken to process a query for the median is $O\left(\log\log N_{max} + \left(1/\varepsilon^2\right)\log(1/\delta)\right)$.

Note that the above guarantees for the sum and the median are only with respect to data that has been received by the aggregator and is within the timestamp window. There may be elements in the stream that have timestamps within the current window, but have not yet arrived at the aggregator, and these are not considered as part of the data on which the sum or the median are computed.

2.1.1.3 Union of Sketches

The sketches produced by our sum and median algorithms can be easily merged to form new sketches. This merging step can be performed repeatedly in a hierarchical manner, using a tree of aggregators. More precisely, given a sketch of stream A and a sketch for stream B, it is easy to obtain a sketch of the union of streams $A \cup B$. A sketch for A (B) consists of a series of random samples from the input stream A (B). The combined sketch consists of a series of random samples from the stream $A \cup B$, which can be computed using the individual random samples from A and B. For the sum,

we show that if each sketch for A and B can individually yield an (ε, δ)-estimator, then the combined sketch can yield an (ε, δ)-estimator for the sum of elements in $A \cup B$. A similar result holds for the median. The space taken for the sketch of the union is no more than the space needed for the sketch of a single stream. Thus, when combining sketches, the new sketch takes bounded space and the relative error is controlled. The cost of transmitting these sketches is small, and this enables the distributed computation of aggregates over the union of many data streams with low communication and space overhead.

2.1.2 Related Work

Datar *et al.* (35) considered basic counting over a sliding window of elements in a data stream under synchronous arrivals. They presented an algorithm that is based on a data structure called the *exponential histogram*, which can give an approximate answer for basic counting, and also presented reductions from other aggregates, such as sum, and ℓ_p norms, to basic counting. For a sliding window size of maximum size W, and an ε relative error, the space taken by their algorithm for basic counting is $O(\frac{1}{\varepsilon}\log^2 W)$, and the time taken to process each element is $O(\log W)$ worst case, but $O(1)$ amortized. Their algorithm for the sum of elements within the sliding window has the space complexity $O(\frac{1}{\varepsilon}\log W(\log W + \log m))$, and worst case time complexity of $O(\log W + \log m)$ where m is an upper bound on the value of an item. We briefly describe the exponential histogram for basic counting. The exponential histogram divides the relevant window of the stream (the last W elements) into buckets of sizes $1, 2, 4, \ldots$. There are multiple buckets of each size (the number of buckets of a particular size depends on the desired accuracy). The most recent elements are grouped into buckets of size 1, elements that arrived a little earlier in time are grouped into buckets of size 2, and even earlier elements are grouped into buckets of size 4, and so on. In a synchronous stream, elements always arrive at in order of timestamps, and hence a newly arrived element is always assigned into a bucket of size 1. This may cause the size of the data structure to exceed the desired maximum, in which case the two least recent buckets of size 1 are merged to form a single bucket of size 2. The merge may cascade, and cause two buckets of size 2 to merge into one bucket of size 4 and so on. This way it is always possible to maintain the invariant that given any large bucket b, there are always many more elements present in buckets that are more recent than b than there are elements in b. In addition, all bucket sizes are powers of

two. In an asynchronous stream, however, the element that just arrived may have an early timestamp. This element may fit into an "old" bucket, causing the size of the bucket to increase, and break the above described invariant. It seems that the exponential histogram is dependent on elements arriving in order of timestamps. Datar *et. al.* (35) also show the following lower bound. If it is assumed that all stream elements have distinct timestamps, then, the space complexity of maintaining an estimate of the sum within an ε relative error (either deterministic or randomized) over a synchronous stream is $\Omega(\log U(\log W + \log U)/\varepsilon)$ bits, where W is the window size and U is an upper bound on the value of an element in the stream. Since a synchronous stream is a special case of an asynchronous stream, this lower bound applies to asynchronous streams too. Under the assumption of distinct timestamps, our algorithm has space complexity $O(\log U(\log W + \log U)/\varepsilon^2)$ for returning an estimate within an ε relative error with a constant probability. This shows that the space cost of asynchrony in this context is no more than $O(1/\varepsilon)$.

Later, Gibbons and Tirthapura (42) gave an algorithm for basic counting based on a data structure called the *wave* that used the same space as in (35), but whose time per element is $O(1)$ worst case. Just like the exponential histogram, the wave also strongly depends on synchronous arrivals, and it does not seem easy to adapt it to the asynchronous case.

Recently, Busch and Tirthapura (14) have devised a deterministic algorithm for estimating the sum (and hence, for basic counting) of elements within a sliding window of an asynchronous stream. Their algorithm has a space complexity of $O(\log U \log W(\log W + \log U)/\varepsilon)$ for returning an answer with ε relative error. When compared with our algorithm for the sum, their algorithm has a worse dependence on $\log W$ and a better dependence on $1/\varepsilon$. Further, their algorithm does not apply to the problem of finding the approximate median.

Arasu and Manku (7) present algorithms to approximate frequency counts and quantiles over a sliding window. Since the median is a special case of a quantile, this also provides a solution for estimating the median, though in the case of synchronous arrivals. Babcock *et al.* (10) presented algorithms for maintaining the variance and k-medians of elements within a sliding window of a data stream. Feigenbaum *et al.* (38) considered the problem of maintaining the diameter of a set of points in the sliding window model.

Gibbons and Tirthapura (41) introduced the distributed streams model. In this model, each of many

distributed parties observes a local stream, has limited workspace, and communicates with a central "referee". When an estimate for the aggregate is requested, the different parties send a "sketch" back to the referee who computes an aggregate over the union of the streams observed by all the parties. In (41), algorithms were presented for estimating the number of distinct elements in the union of distributed streams, and the size of the bitwise-union of distributed streams. In a later work (42), they considered estimation of functions over a sliding window on distributed streams. However, the algorithms in (42) were designed for the case of synchronous arrivals. Patt-Shamir (66) presented communication efficient algorithms for computing various aggregates, such as the median and number of distinct elements in a sensor network, and considered multi-round distributed algorithms for that purpose.

Guha, Gunopulos, and Koudas (45) consider the problem of computing correlations between multiple vectors. The vectors arrive as multiple data streams, and within each stream, the elements of a vector arrive as updates to existing values; the updates are asynchronous, and do not necessarily arrive in order of the indexes of elements. Their work focuses on the approximate computation of the largest eigenvalues of the resulting matrix, using limited space and in one pass on both synchronous and asynchronous data streams. They do not consider the context of sliding windows.

Srivastava and Widom (72) designed a *heartbeat* generation algorithm to support continuous queries in a Data Stream Management System, which receives multiple asynchronous data streams. Each stream is a sequence of tuples of the form $\langle value, timestamp \rangle$. The timestamp is tagged by the source of the stream. By capturing the skew between steams, and the asynchrony and network transmission latency of each stream, their algorithm can generate and update a "heartbeat" continuously. The algorithm guarantees that there will be no new tuples arriving with a timestamp earlier than the heartbeat. All tuples with timestamp greater than the current heartbeat are buffered. Once the heartbeat is updated (advanced), all buffered tuples with timestamp earlier than the new heartbeat are submitted to the query processor to answer continuous queries. Their algorithm requires that the skew between streams, and the asynchrony and the network transmission latency of each stream be bounded, while our algorithm works on *any* asynchronous stream. Their work does not consider maintenance of aggregates, as we do here.

Another sketch that is popular in networking applications is the Bloom filter (11), which summarizes a set of items to support approximate membership queries. A Bloom filter tackles a different type

of sketching problem than we do – our sketches are designed to support aggregate queries on data, while a Bloom filter supports queries about the existence (or not) of individual elements in the data. Since keeping information about individual elements is clearly expensive, a Bloom filter is a rather bulky sketch when compared to the sketches we present here. The space taken by our sketches do not depend on the number of elements in the data set (it only depends on the desired accuracy), while the size of a Bloom filter is linear in the number of elements.

Much other recent work on data stream algorithms has been surveyed in (8; 62). To our knowledge, our work is the first to consider aggregates over sliding windows under asynchronous arrivals.

2.2 Sum of Positive Integers

We first consider the computation of the sum in the centralized model. The stream received by the aggregator is $R = \langle d_1 = (v_1,t_1), d_2 = (v_2,t_2), \ldots, d_n = (v_n,t_n)\rangle$ where the v_is are positive integers and t_is are the timestamps. Recall c denotes the current time at the aggregator. The goal is to maintain a sketch of the stream R which will provide an answer for the following query. *For a user provided w that is given at the time of the query, what is the sum of the observations within the current timestamp window $[c-w,c]$?* The sketch should be quickly updated as new elements arrive, and no assumptions can be made on the order of arrivals.

We assume that the algorithm knows W, an upper bound on the window size. For window size $w \leq W$, let R_w denote the set of observations within the current timestamp window, i.e. $R_w = \{(v,t) \in R | c-w \leq t \leq c\}$. Given w, the sketch should return an estimate of V, the sum of input observations within R_w. $V = \sum_{\{(v,t)\in R_w\}} v$

The value of W depends on the application. For example, in a network monitoring application, the user (network administrator) may never have an interest in querying about packets that were generated more than 24 hours ago, in which case setting W to be 24 hours will suffice. Note than W can also be set to infinity, which essentially means that the sketch summarizes the whole stream.

2.2.1 Intuition

Our algorithm is based on *random sampling*. The high level idea is as follows. In order to estimate the sum of integers within the sliding window, the stream elements are randomly chosen into a sample

as they are observed by the aggregator. When an estimate is asked for the sum of elements in a given timestamp window, the algorithm computes the sum of all elements in the sample that are within the timestamp window, multiplies it by the appropriate factor (inverse of the sampling probability), and returns the product as the estimate. The description thus far is the recipe for most estimation algorithms that are based on random sampling. In getting random sampling to work for this scenario, we need the following ideas.

First, suppose the goal is to estimate the cardinality of a set using random sampling. In order to get a desired accuracy for the estimate, it is enough to sample the elements of the set such that the size of the resulting sample is "large enough"; what is "large enough" depends only on the desired accuracy (ε and δ), and not on the size of the set itself. The required size of the sample can be determined using Chernoff bounds.

Next, in estimating the sum, different elements in the stream have to be treated with different weights during random sampling, otherwise the error in estimation could become too large. For example, two observations $d_1 = (100, t)$ and $d_2 = (1, t)$ may both be included in the current sliding timestamp window, but the sampling should give greater weight to d_1 than to d_2, to maintain a good accuracy for the estimate. If every element is sampled with the same probability, it can be verified that the expected value of the estimate is correct, but the variance of the estimate is too large for our purposes. The exact differences in the handling of elements with different values is crucial for guaranteeing the error bounds, and for further details on this we refer the reader to the formal description of the algorithm. We note that many of the technical proofs in this chapter are devoted to this aspect of handling elements with varying weights.

Finally, the "correct" probability of sampling cannot be predicted before the query for the sum is asked. If the answer for the sum is large (estimation of the size of a "dense" set), then a small sampling probability may be enough to return an accurate estimate. If the answer for the sum is small (estimation of the size of a "sparse" set), then a larger sampling probability may be necessary. Thus, our algorithm maintains not just one random sample, but many random samples, at probabilities $p = 1, 1/2, 1/4, \ldots$ Clearly, the samples at larger probabilities may be too large to fit within the workspace, but we show that in each sample, it suffices to maintain only the most recent elements selected into the sample. When a query is asked, with high probability, one of these samples will provide a good estimate for the

Algorithm 1: SumInit()

Task: Initialize the sketch.

```
1 for i = 0...M do
2     S_i ← φ;                          /* All samples initially empty */
3     t_i ← −1 ;                        /* No items have been discarded yet */
```

sum of all elements within the sliding timestamp window. In our actual algorithm however, all samples are not explicitly stored. To improve the element processing time, each element is stored only in the lowest probability sample that it is selected into. When required to answer a query for the sum, the required sample is reconstructed using all samples at lower probabilities.

2.2.2 Formal Description of the Algorithm

We assume that the algorithm knows an upper bound V_{max} on the value of V. The space complexity of the sketch depends on $\log V_{max}$. For example, if an upper bound m was known on each value v corresponding to the sum of elements at a time instant, and there were no more than f stream elements with the same timestamp, then mfW is a trivial upper bound on V.

Let $M = \lceil \log V_{max} \rceil$. The algorithm maintains $(M+1)$ samples, denoted $S_0, S_1, \ldots, S_M$. Sample S_i is said to be at "level" i. Each sample S_i contains the most recent elements selected into the sample, and when more elements enter the sample, older elements are discarded. Let t_i be the most recent timestamp of elements discarded from S_i. The purpose of t_i is to help in determining the range of timestamps that are still present in the sample. The maximum number of elements in each sample S_i is $\alpha = (12/\varepsilon^2)\ln(8/\delta)$.

The algorithm is described in algorithms *SumInit*, which describes the initialization steps for the sketch, *SumProcess* which describes the algorithm for updating the sketch upon receiving a new element, and *SumQuery*, which describes the steps for answering a query for the sum.

2.2.3 Correctness Proof

Let X denote the result returned by *SumQuery(w)* when a query is asked for the sum of elements within the sliding timestamp window $[c-w, c]$. We show that X is an (ε, δ)-estimate of V.

Algorithm 2: SumProcess(d=(v,t))

Input: v is the value of the element, and is a positive integer; t is the timestamp
Task: Insert d into the sketch.

if $(t < c - W)$ **then return** ; `/* Discard d since it is outside the largest timestamp window, and a future query will never involve d. */`
Let $\ell = \min\left\{i | 0 \le i \le M, v/2^\ell < 1\right\}$; `/* ℓ is an integer. */`
Let $\Pr[r=1] = v/2^\ell$ and $\Pr[r=0] = 1 - v/2^\ell$;
if $r = 1$ **then** $k \leftarrow \min\{Z, M - \ell + 1\}$, where Z is the number of flips of a fair coin till the first tail;
if $r = 0$ **then** $k \leftarrow 0$;
Insert (v,t) into $S_{\ell+k-1}$;
if $|S_{\ell+k-1}| > \alpha$ **then**
    Discard the element with the lowest timestamp in $S_{\ell+k-1}$;
    Let t' be the timestamp of the discarded element;
    $t_{\ell+k-1} \leftarrow \max\{t_{\ell+k-1}, t'\}$;

Algorithm 3: SumQuery(w)

Input: $w \le W$ is the width of the window
Output: An estimate of the sum of all stream elements with timestamps in the range $[c-w, c]$

Let $\ell' \in [0, M]$ be the smallest integer, such that for all $\ell' \le j \le M$, $t_j < c - w$;
if *no such* ℓ' *exists* **then** $\ell' \leftarrow M + 1$;
if $\ell' \le M$ **then**
    for $i = \ell'$ *to* M **do** $\eta_i \leftarrow \sum_{(v,t) \in S_i, t \ge c-w} \max(\frac{v}{2^{\ell'}}, 1)$;
    return $2^{\ell'} \sum_{i=\ell'}^{M} \eta_i$;
if $\ell' = M + 1$ **then return** ; `/* Algorithm Fails */`

Definition 2.2.1. *For each element $d = (v,t) \in R_w$, for each level $i = 0, 1, 2, \ldots, M$, random variable $x_i(d)$ is defined as follows. Let γ be the smallest level such that $v/2^\gamma < 1$.*

- *For $0 \le i < \gamma$, $x_i(d) = v/2^i$.*
- *$x_\gamma(d) = 1$ with probability $v/2^\gamma$, and $x_\gamma(d) = 0$ with probability $1 - v/2^\gamma$.*
- *For $\gamma < i \le M$, $x_i(d)$ is defined inductively. If $x_{i-1}(d) = 0$ then, $x_i(d) = 0$. If $x_{i-1}(d) = 1$, then $x_i(d) = 1$ with probability $\frac{1}{2}$ and $x_i(d) = 0$ with probability $1/2$.*

Definition 2.2.2. *For $i = 0, \ldots, M$, T_i is the set constructed by the following probabilistic process. Start with $T_i \leftarrow \phi$. For each element $d = (v,t) \in R_w$, if $x_i(d) \ne 0$, then insert $(x_i(d), t)$ 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.

Definition 2.2.3. *For $i = 0, \ldots, M$, define $X_i = \sum_{(u,t)\in T_i} u$.*

Lemma 2.2.1. *If $d = (v,t)$ then $E[x_i(d)] = v/2^i$*

Proof. Let γ be defined as in Definition 2.2.1, i.e. γ is the smallest level such that $\frac{v}{2^\gamma} < 1$. For $0 \le i < \gamma$ $E[x_i(d)] = v/2^i$, since $x_i(d)$ is a constant. For $\gamma \le i \le M$, $x_i(d)$ is a 0-1 random variable. We use proof by induction on i to show that $E[x_i(d)] = \Pr[x_i(d) = 1] = v/2^i$. The base case $i = \gamma$ is true since $\Pr[x_\gamma(d) = 1] = v/2^\gamma$ by definition. Assume that for $i \ge \gamma$, $\Pr[x_i(d) = 1] = v/2^i$. Again, using Definition 2.2.1, $\Pr[x_{i+1}(d) = 1] = (1/2) \cdot \Pr[x_i(d) = 1] = v/2^{i+1}$, thus proving the inductive step. □

Lemma 2.2.2. *For $i = 0, \ldots, M$, $E[X_i] = V/2^i$*

Proof. The definitions of X_i and $x_i(d)$ yield the following.

$$X_i = \sum_{(u,t)\in T_i} u = \sum_{d=(v,t)\in R_w} x_i(d)$$

Using linearity of expectation and Lemma 2.2.1, we get:

$$E[X_i] = \sum_{d=(v,t)\in R_w} E[x_i(d)] = \sum_{d=(v,t)\in R_w} \frac{v}{2^i} = \frac{1}{2^i} \sum_{d=(v,t)\in R_w} v = \frac{V}{2^i}$$

□

Lemma 2.2.3. *When asked for an estimate for V, if* SumQuery(w) *does not fail in Step 6, then it returns $2^{\ell'} X_{\ell'}$ for value ℓ' selected in Step 1.*

Proof. Consider *SumQuery(w)* when asked for an estimate of the sum of elements in R_w.

Note that the level chosen by the algorithm, ℓ', satisfies the following condition. For all levels $\ell' \le i \le M$, the most recent timestamp of the discarded elements (contained in the variable t_i in the algorithm) is less than $c - w$. Thus, for all $i, \ell' \le i \le M$, no element which is selected into S_i and has a timestamp at least $c - w$ is discarded.

Next, we argue that the contribution of each element $d = (v,t) \in R_w$ to the value returned by the algorithm is $2^{\ell'} x_{\ell'}(d)$. Suppose $x_{\ell'}(d) = 0$. We refer to the algorithm for processing an element in *SumProcess(d)*. The arrival of element $d = (v,t)$ causes an insertion of (v,t) into S_i for level $i < \ell'$. Note

that in computing the estimate, *SumQuery(w)* only uses elements from levels ℓ' or greater, element d will not contribute to the estimate returned by the algorithm.

Suppose $x_{\ell'}(d) > 0$. Again, referring to *SumProcess(d)*, we note that the arrival of d causes an insertion of (v,t) into a level $i \geq \ell'$. In answering a query for the sum (*SumQuery(w)*), all elements with timestamp at least $c-w$ which are inserted into levels ℓ' or greater are considered, and their contribution to the estimate is exactly $2^{\ell'}x_{\ell'}(d)$. To see this, suppose $v \geq 2^{\ell'}$. Then, $x_{\ell'}(d) = v/2^{\ell'}$. From Step 4 in *SumQuery(w)*, the contribution of v to the estimate is $2^{\ell'}(v/2^{\ell'}) = 2^{\ell'}x_{\ell'}(d)$. Suppose $v < 2^{\ell'}$. Then $x_{\ell'}(d)$ should be 1, since it is a 0-1 random variable. In such a case, from Step 4 in *SumQuery(w)*, the contribution of d to the estimate is $2^{\ell'} = 2^{\ell'}x_{\ell'}(d)$. Thus, for each $d = (v,t) \in R_w$, the contribution to the returned estimate is $2^{\ell'}x_{\ell'}(d)$. The total returned estimate is exactly $2^{\ell'}X_{\ell'}$. □

Next, we will show that $X_{\ell'}$ is a good estimate for V. The following definition captures the notion of whether or not different samples yield good estimates for V.

Definition 2.2.4. *For $i = 0,\ldots,M$, random variable X_i is said to be "good" if $(1-\varepsilon)V \leq X_i 2^i \leq (1+\varepsilon)V$, and "bad" otherwise. Define event B_i to be true if X_i is bad, and false otherwise.*

Lemma 2.2.4. *If $|R_w| \leq \alpha$, then* SumQuery(*w*) *returns the exact answer for the sum.*

Proof. Note that each element in R_w was selected into S_0 when it was processed. Since the α elements with the most recent timestamps are stored in S_0, it must be true that $R_w \subseteq S_0$. *SumQuery* will retrieve all of R_w from S_0 and return the exact sum of R_w. □

Because of the above lemma, in the rest of the proof, we assume $|R_w| > \alpha$. Since each element in the input stream is at least 1, this implies that $V > \alpha$.

Definition 2.2.5. *Let $\ell^\star \geq 0$ be an integer such that $E[X_{\ell^\star}] \leq \alpha/2$ and $E[X_{\ell^\star}] > \alpha/4$.*

Lemma 2.2.5. *Level $\ell^\star$ is uniquely defined and exists for every input stream R.*

Proof. From Lemma 2.2.2, we have $E[X_i] = V/2^i$. Since $V > \alpha$, $E[X_0] > \alpha$. By the definition of $M = \lceil \log V_{max} \rceil$, it must be true that $V \leq 2^M$ for any input stream R, so that $E[X_M] \leq 1$. Since for every increment in i, $E[X_i]$ decreases by a factor of 2, there must be a unique level $0 < \ell^\star < M$ such that $E[X_{\ell^\star}] \leq \alpha/2$ and $E[X_{\ell^\star}] > \alpha/4$. □

For the next lemmas, we use a version of Hoeffding bounds from Schmidt, Siegel and Srinivasan (70) (Section 2.1) which is restated here for convenience. Let $y_1, y_2, \ldots,$ y_n be independent 0-1 random variables with $\Pr[y_i = 1] = p_i$. Let $Y = y_1 + y_2 + \ldots + y_n$, and let $\mu = E[Y]$.

Lemma 2.2.6. Hoeffding's Bound (restated from (70)):

(1) If $0 < \delta < 1$, *then* $\Pr[Y > \mu(1+\delta)] \leq e^{-\mu\delta^2/3}$.

(2) If $\delta \geq 1$, *then* $\Pr[Y > \mu(1+\delta)] \leq e^{-\mu\delta/3}$.

(3) If $0 < \delta < 1$, *then* $\Pr[Y < \mu(1-\delta)] \leq e^{-\mu\delta^2/2}$.

The next lemma helps in the proof of Lemma 2.2.8.

Lemma 2.2.7. *If* $0 < a < \frac{1}{2}$ *and* $k \geq 0$, *then* $a^{(2^k)} \leq \frac{a}{2^k}$

Proof. It is clear by induction that $2^k - 1 \geq k$. Since $0 < a < \frac{1}{2}$, we can further have $a^{(2^k-1)} \leq a^k < (1/2)^k$. Therefore, $a^{(2^k)} < \frac{a}{2^k}$. □

The next lemma shows that it is highly unlikely that B_ℓ is true for any ℓ such that $0 \leq \ell \leq \ell^\star$.

Lemma 2.2.8. *For integer* ℓ *such that* $0 \leq \ell \leq \ell^\star$,

$$\Pr[X_\ell \notin (1-\varepsilon, 1+\varepsilon)E[X_\ell]] < \frac{\delta}{2^{\ell^\star - \ell + 2}}$$

Proof.

$$X_\ell = \sum_{d=(v,t)\in R_w} x_\ell(d)$$

From Definition 2.2.1, it follows that for some $d \in R_w$, $x_\ell(d)$ is a constant and for others $x_\ell(d)$ is a 0-1 random variable. Thus, X_ℓ is the sum of a few constants and a few random variables. Let $X_\ell = c + Y$ where c denotes the sum of all $x_\ell(d)$'s that are constants, and Y is the sum of the $x_\ell(d)$'s that are 0-1 random variables. Clearly, since the different elements of the stream are sampled using independent random bits, the random variables $x_\ell(d)$ for different $d \in R_w$ are all independent. Thus Y is the sum of independent 0-1 random variables. Let $\mu_Y = E[Y]$.

By linearity of expectation, we have

$$E[X_\ell] = c + \mu_Y \tag{2.1}$$

By the definition of $\ell^\star$, $E[X_{\ell^\star}] > \alpha/4$. Since $E[X_i] = \frac{V}{2^i}$ (from Lemma 2.2.2). Using Equation 2.1, we get the following inequality that will be used in further proofs.

$$c + \mu_Y > 2^{\ell^\star - \ell}(\alpha/4) \tag{2.2}$$

We first consider $\Pr[X_\ell > (1+\varepsilon)E[X_\ell]]$

$$\begin{aligned}
\Pr[X_\ell > (1+\varepsilon)E[X_\ell]] &= \Pr[c+Y > (1+\varepsilon)(c+\mu_Y)] \\
&= \Pr[Y > \mu_Y(1+\frac{\varepsilon(c+\mu_Y)}{\mu_Y})] \\
&= \Pr[Y > \mu_Y(1+\delta')],
\end{aligned}$$

Where $\delta' = \varepsilon(c+\mu_Y)/\mu_Y$.

We consider two cases here: $\delta' < 1$ and $\delta' \geq 1$.

Case I: $\delta' < 1$**.** Using Lemma 2.2.6 and the fact $(c+\mu_Y)/\mu_Y \geq 1$, we have

$$\begin{aligned}
\Pr[Y > \mu_Y(1+\delta')] &\leq e^{-\mu_Y \delta'^2/3} = e^{-\frac{\varepsilon^2(c+\mu_Y)^2}{3\mu_Y}} \\
&\leq e^{-\varepsilon^2(c+\mu_Y)/3} < e^{-\varepsilon^2(2^{\ell^\star-\ell}(\alpha/4))/3} \\
&< \left(\frac{\delta}{8}\right)^{(2^{\ell^\star-\ell})} \leq \frac{\delta/8}{2^{\ell^\star-\ell}}
\end{aligned}$$

Where we have used $\alpha = (12/\varepsilon^2)\ln(8/\delta)$ and $\delta < 1$, Equation 2.2 and Lemma 2.2.7.

Case II: $\delta' \geq 1$**.** Using Lemma 2.2.6, we have:

$$\begin{aligned}
\Pr[Y > \mu_Y(1+\delta')] &\leq e^{-\mu_Y \delta'/3} = e^{-\varepsilon(c+\mu_Y)/3} \\
&< e^{-2^{(\ell^\star-\ell)}\ln(8/\delta)/\varepsilon} = [(\frac{\delta}{8})^{1/\varepsilon}]^{2^{\ell^\star-\ell}} \\
&< (\frac{\delta}{8})^{(2^{\ell^\star-\ell})} \leq \frac{\delta/8}{2^{\ell^\star-\ell}}
\end{aligned}$$

where we have used $\alpha = (12/\varepsilon^2)\ln(8/\delta)$ and $\delta < 1$, Equation 2.2 and Lemma 2.2.7.

From Case I and Case II, we have

$$\Pr[X_\ell < (1+\varepsilon)E[X_\ell]] < \frac{\delta/8}{2^{\ell^\star-\ell}} \tag{2.3}$$

Next we consider $\Pr[X_\ell < (1-\varepsilon)E[X_\ell]]$

$$\Pr[X_\ell < (1-\varepsilon)E[X_\ell]] = \Pr[c+Y < (1-\varepsilon)(c+\mu_Y)] = \Pr[Y < \mu_Y(1-\delta')]$$

Where $\delta' = \varepsilon(c+\mu_Y)/\mu_Y$

Using Lemma 2.2.6 and the fact $(\mu_Y + c)/\mu_Y \geq 1$,

$$\Pr[Y < \mu_Y(1-\delta')] \leq e^{-\mu_Y\delta'^2/2} = e^{-\frac{\varepsilon^2(\mu_Y+c)^2}{2\mu_Y}} \leq e^{-\varepsilon^2(\mu_Y+c)/2}$$

Using Equation 2.2 and Lemma 2.2.7,

$$e^{-\varepsilon^2(\mu_Y+c)/2} < [(\frac{\delta}{8})^{\frac{3}{2}}]^{(2^{\ell^\star-\ell})} < (\frac{\delta}{8})^{(2^{\ell^\star-\ell})} \leq \frac{\delta/8}{2^{\ell^\star-\ell}}$$

Thus, we have

$$\Pr[X_\ell < (1-\varepsilon)E[X_\ell]] < \frac{\delta/8}{2^{\ell^\star-\ell}} \tag{2.4}$$

Combining Equations 2.3 and 2.4, for $0 \leq \ell \leq \ell^\star$, we get

$$\Pr[X_\ell \notin (1-\varepsilon, 1+\varepsilon)E[X_\ell]] = \Pr[X_\ell > (1+\varepsilon)E[X_\ell]] + \Pr[X_\ell < (1-\varepsilon)E[X_\ell]] < \frac{\delta/4}{2^{\ell^\star-\ell}}$$

□

Lemma 2.2.9.

$$\sum_{i=0}^{\ell^\star} \Pr[B_i] < \delta/2$$

Proof. By definition of B_i, $\Pr[B_i] = \Pr[2^i X_i \notin (1-\varepsilon, 1+\varepsilon)V] = Pr[X_i \notin (1-\varepsilon, 1+\varepsilon)E[X_i]]$

Using Lemma 2.2.8,

$$\sum_{i=0}^{\ell^\star} \Pr[B_i] < \sum_{i=0}^{\ell^\star} \frac{\delta}{2^{\ell^\star-i+2}} = \frac{\delta}{4}\sum_{j=0}^{\ell^\star}\frac{1}{2^j} < \delta/2$$

□

Recall that the algorithm uses level ℓ' in *SumQuery(w)* to answer the query for the sum.

Lemma 2.2.10.

$$\Pr[\ell' > \ell^\star] < \delta/8$$

Proof. Let β_i denote the number of elements of R_w that were inserted into S_i. By *SumQuery(w)*, we know that $\beta_{\ell'-1} > \alpha$ since otherwise the algorithm would have chosen level $\ell'-1$ instead. Note that for each level $i = 0 \ldots M$, $|T_i| \geq \beta_i$, since an insertion of an element in R_w into S_i always causes an insertion into T_i (but not necessarily vice versa). Thus, $|T_{\ell'-1}| > \alpha$. Note that from Definition 2.2.2, it follows that for all $0 \leq i_1 < i_2 \leq M$, $|T_{i_1}| \geq |T_{i_2}|$. Thus, if $\ell' - 1 \geq \ell^\star$, then $|T_{\ell^\star}| \geq |T_{\ell'-1}| > \alpha$.

$$\Pr[\ell' > \ell^\star] = \Pr[\ell' - 1 \geq \ell^\star] \leq \Pr[|T_{\ell^\star}| > \alpha] \tag{2.5}$$

Since each element in T_i contributes at least one to X_i, we have $X_i \geq |T_i|$. Combining this with Equation 2.5, we get:

$$\Pr[\ell' > \ell^\star] \leq \Pr[X_{\ell^\star} > \alpha] \tag{2.6}$$

As in the proof of Lemma 2.2.8, we denote $X_{\ell^\star} = c + Y$, where c is a constant and Y is the sum of independent 0-1 random variables. Let $\mu_Y = E[Y]$. Since $E[X_{\ell^\star}] \leq \alpha/2$, we have

$$\begin{aligned} \Pr[X_{\ell^\star} > \alpha] &\leq \Pr[X_{\ell^\star} > 2E[X_{\ell^\star}]] \\ &= \Pr[c + Y > 2(c + \mu_Y)] \\ &= \Pr[Y > \mu_Y(1 + \frac{c + \mu_Y}{\mu_Y})] \end{aligned}$$

Using Lemma 2.2.6,

$$\Pr[Y > \mu_Y(1 + \frac{c + \mu_Y}{\mu_Y})] < e^{-\mu_Y \delta'/3} = e^{-(c+\mu_Y)/3}$$

Where $\delta' = (c + \mu_Y)/\mu_Y > 1$.

Since, $E[X_{\ell^\star}] = c + \mu_Y > \alpha/4$, we have

$$e^{-(c+\mu_Y)/3} < e^{-\frac{\ln(8/\delta)}{\varepsilon^2}} = (\frac{\delta}{8})^{1/\varepsilon^2} < \frac{\delta}{8}$$

□

Theorem 2.2.1. *The result of the algorithm, $X_{\ell'}$, is an (ε,δ)-estimate for V, the sum of all elements in the timestamp window $[c-w,c]$.*

Proof. Let f denote the probability that the algorithm fails to return an estimate that is within an ε relative error of V. Note that one way the algorithm can fail is by running out of levels, i.e. at level M the sample still has too many elements; as we show, this is an unlikely event.

$$\begin{aligned}
f &= \Pr[\ell' > M] + \Pr[\bigcup_{i=0}^{M} (\ell' = i) \wedge B_i] \\
&\leq \Pr[\ell' > M] + \sum_{i=0}^{M} \Pr[(\ell' = i) \wedge B_i] \\
&\leq \Pr[\ell' > M] + \sum_{i=0}^{\ell^\star} \Pr[B_i] + \sum_{i=\ell^\star+1}^{M} \Pr[\ell' = i] \\
&= \Pr[\ell' > \ell^\star] + \sum_{i=0}^{\ell^\star} \Pr[B_i] \\
&< \frac{\delta}{8} + \frac{\delta}{2} \\
&< \delta
\end{aligned}$$

where we have used Lemmas 2.2.9 and 2.2.10. □

2.2.4 Complexity

Lemma 2.2.11. Space Complexity: *The total space taken by the sketch for the sum is $O\left((1/\varepsilon^2)\,(\log(1/\delta))(\log V_{max})\sigma\right)$, where V_{max} is an upper bound on the value of the sum V, σ is the space taken to store an input element (v,t), ε is the desired relative error, and δ is the desired upper bound on the failure probability.*

Proof. The algorithm maintains $M = \lceil \log V_{max} \rceil$ samples, each of which has up to $\alpha = (12/\varepsilon^2)\ln(8/\delta)$ elements. Each element in the sample is a pair (v,t), which can be stored using σ bits. The product of

the number of samples, the number of elements per sample, and the space per element yields the above space complexity. □

Lemma 2.2.12. Time Complexity: *The worst case time complexity for processing an element* $d = (v,t)$ *by* SumProcess(d) *is* $O(\log \alpha) = O(\log\log(1/\delta) + \log(1/\varepsilon))$. *The worst case time taken to answer a query for the sum by* SumQuery(w) *is* $O(M\alpha) = O((1/\varepsilon^2) \cdot \log V_{max} \cdot \log(1/\delta))$.

Proof. The elements in each sample can be stored using a heap that is ordered according to the timestamps of the elements. The heap supports two operations, (a)insertion and (b)delete-min, both in time $O(\log \alpha)$, since the maximum size of each sample is $\alpha = (12/\varepsilon^2)\ln(8/\delta)$.

Consider *SumProcess(d)*. If input element $d = (v,t)$ is outside the window (Step 1), then it takes constant time to discard it. Otherwise, the time taken to process d consists of three parts. The first part is to compute the value of ℓ in Step 2, which takes constant time. The second part is to find the value of k in Step 4 of *SumProcess(d)*. We assume that it takes constant time to generate an exponentially distributed random number k, where $\Pr[k = i] = 1/2^i$, $i = 1, 2, \cdots$. Thus, Step 4 also takes constant time. The third part is the actual insertion into $S_{\ell+k-1}$ in Step 6, and (possibly) discarding the oldest element of $S_{\ell+k-1}$ in Step 7, which takes $O(\log \alpha)$ time. Summing these, we find that the worst case time to process d is $O(\log \alpha)$.

The time taken to answer a query for the sum consists of two parts. The first part is to find the value of ℓ in *SumQuery(w)*, which can be done in $O(M)$ time. The second part is to find all elements with timestamps within the query window in sample S_i, $\ell \leq i \leq M$. This part takes time $O(M\alpha)$. Summing these two parts, the worst case time taken for answering a query is $O(M\alpha)$. □

2.2.5 Trade off between Processing time and Query time.

By spending more time during processing an element, it is possible to improve the query time for the sum as follows. In algorithm *SumProcess(d)*, instead of inserting the element into only one level $(\ell + k - 1)$ in Step 6, it can be inserted into every level starting from 0 till $(\ell + k - 1)$ (Figure 2 of (75)). This way, when processing a query for the sum in *SumQuery*, we need to consider only a single level ℓ' (Figure 3 of (75)), rather than all levels from ℓ' till M. The space complexity of the algorithm would remain the same as before, but the time complexity would change as follows. Worst case time for

processing an element is now $O((\log V_{max})(\log\log 1/\delta + \log 1/\varepsilon))$, and time taken to answer a query for the sum is $O(\log\log V_{max} + \log(1/\delta)/\varepsilon^2)$. The time for answering the query has decreased, while the time for processing an element has increased. In most applications, since queries are likely to be much less frequent than element arrivals, the algorithm with faster element processing time may be preferred (i.e. algorithms *SumProcess* and *SumQuery*).

A more flexible trade off between processing time and query time can be obtained as follows. The user can specify a level L, $0 \leq L < M$, as a parameter to *SumProcess* and *SumQuery*. In *SumProcess*, if $(\ell + k - 1) < L$, then insert the element into only one level $(\ell + k - 1)$; otherwise, insert the element into levels $L, L+1, \ldots, \ell + k - 1$. *SumQuery* is modified as follows. As before, level $\ell' \in [0, M]$ is the smallest integer such that for all j, $\ell' \leq j \leq M$, $t_j < c - w$. If $\ell' < L$, then the query is answered using the union of all elements in levels $\ell'+1, \ell'+2, \ldots, L$ that belong within the window. On the other hand, if $\ell' \geq L$, then the query is answered using only the elements in level ℓ', since the relevant elements in later levels are also present in level ℓ'.

With this modification, the space complexity remains the same as before, but the time complexity changes as follows. Worst case time for processing an element is now $O\left((\lceil \log V_{max} \rceil - L)(\log\log(1/\delta) + \log(1/\varepsilon))\right)$, and worst case time for answering a query for the sum is $max\left(O\left(\log(\lceil \log V_{max} \rceil - L) + (\log(1/\delta))/\varepsilon^2\right), O\left(L \cdot (\log(1/\delta))/\varepsilon^2\right)\right)$. The smaller the value of L is, the more time spent on processing an element, but the less time spent on answering a query, and vice versa. Clearly if we choose $L = 0$, the algorithm for processing an element is the one in Figure 2 of (75), i.e., the element will be inserted into every level that it is selected into, and the algorithm to answer a query for the sum is the one in Figure 3 of (75); if we choose $L = M - 1$, it is *SumProcess* in Section 2.2.2 which process an element, and the algorithm for answering a query for the sum is *SumQuery* in Section 2.2.2.

2.3 Computing the Median

In this section, given a maximum window size W, we design a sketch such that for all $w \leq W$, the sketch can return an (ε, δ)-approximate median of R_w, whose values are chosen from a totally ordered universe.

The algorithm for the median is based on random sampling, as are many earlier algorithms for

medians and quantiles over data streams (58; 44). Roughly speaking, the median of a random sample of a stream, where the stream is sampled at a sufficiently large probability, is an approximate median of all elements in the stream. What is a "sufficiently large" probability depends on the size of the set on which the median is being computed, and the desired accuracy. Since the window size w is known only at query time, there is no single sampling probability that suffices for all queries. Similar to the algorithm for the sum, the idea in the algorithm for the median is to maintain many random samples at different probabilities, starting with a probability of 1 and successively decreasing by a factor of $1/2$. The key differences between the algorithms for the sum and median are summarized below – though these algorithms are similar from a high level, these differences make the correctness proofs quite different.

1. In the algorithm for the median, the value of the data item does not affect the sampling probability. A uniform random sample suffices for the median, while a non-uniform sample is necessary for the sum.

2. Another simplification in the sketch for the median is that each element is explicitly stored in every level that it is sampled into. In the case of the median, storing an element explicitly in each level is not expensive, since on average, each element is sampled into only two levels. Storing the element in this way improves the cost of a query for the approximate median, while it does not significantly alter the cost of processing an element. In the case of the sum, however, storing the element explicitly in each level it is sampled may be expensive, since an element with a high value will be sampled into many levels.

2.3.1 Formal Description of the Algorithm

We assume that the algorithm knows an upper bound N_{max} on the number of elements in R_w. For example, if there were no more than f elements with the same timestamp then setting $N_{max} = fW$ will do. The space complexity of the sketch depends on $\log N_{max}$. Let $N = |R_w|$, $M = \lceil \log N_{max} \rceil$.

The algorithm for the median maintains $(M+1)$ samples $S_0, S_1, \ldots, S_M$ and the corresponding t_i's. The maximum number of elements in each sample S_i is $\alpha = (96/\varepsilon^2)\ln(8/\delta)$. Initially, each S_i is empty and t_i is set to be -1, as described in *SumInit*. The algorithm for updating the sketch upon receiving

Algorithm 4: MedianProcess(d=(v,t))

Input: v is the value of the element, and is a positive integer; t is the timestamp
Task: Insert d into the sketch.

```
1  if (t < c − W) then return;
2  Insert (v,t) into S_0;
3  if |S_0| > α then
4      Discard the element with the earliest timestamp in S_0, say t′;
5      Update t_0 ← max{t_0, t′};
6  Set i ← 1;
7  while (v,t) was inserted into level (i − 1) and i ≤ M do
8      Insert (v,t) into S_i with probability 1/2;
9      if |S_i| > α then
10         Discard the element with the earliest timestamp in S_i, say t′;
11         Update t_i ← max{t_i, t′};
12     Increment i;
```

Algorithm 5: MedianQuery(w)

Input: $w \leq W$ is the width of the window
Output: An estimate of the median of all stream elements with timestamps in the range $[c-w, c]$

```
1  Let ℓ′ be the smallest integer 0 ≤ ℓ′ ≤ M such that t_ℓ′ < c − w;
2  if ℓ′ exists then return the median of the set {(v,t) ∈ S_ℓ′ | t ≥ c − w};
3  else ℓ′ ← M + 1;
4  if ℓ′ = M + 1 then return ;                          /* Algorithm fails */
```

a new element is described in *MedianProcess*, and *MedianQuery* returns an estimate of the median of R_w when receiving a query.

2.3.2 Correctness Proof

We now show that the result of *MedianQuery(w)* is an (ε, δ)-approximate median of the set R_w.

Definition 2.3.1. *For each element $d = (v,t) \in R_w$, for each level $i = 0, 1, 2, \ldots, M$, random variable $x_i(d)$ is defined inductively as follows.*

- $x_0(d) = 1$
- *For $i > 0$, if $x_{i-1}(d) = 1$, then $x_i(d) = 1$ with probability $\frac{1}{2}$ and $x_i(d) = 0$ with probability $1/2$. If $x_{i-1}(d) = 0$, then $x_i(d) = 0$.*

Definition 2.3.2. *For $i = 0, 1, \ldots, M$, T_i is the set constructed by the following probabilistic process. Start with $T_i \leftarrow \phi$. For each element $d = (v,t) \in R_w$, if $x_i(d) = 1$, then insert (v,t) into T_i. Let $X_i = |T_i|$.*

Lemma 2.3.1. *Given any $d = (v,t)$, for each i, $0 \leq i \leq M$, $E[x_i(d)] = 1/2^i$, $E[X_i] = N/2^i$.*

Proof. We use proof by induction on i to show that $E[x_i(d)] = \Pr[x_i(d) = 1] = 1/2^i$. The base case $i = 0$ is true by Definition 2.3.1. Assume for $0 \leq i < M$, $\Pr[x_i(d) = 1] = 1/2^i$. Using Definition 2.3.1, $\Pr[x_{i+1}(d) = 1] = 1/2 \cdot \Pr[x_i(d) = 1] = 1/2^{i+1}$, proving the inductive step.

Now we show $E[X_i] = N/2^i$. Note that $|R_w| = N$. The Definitions 2.3.1 and 2.3.2 yield $X_i = |T_i| = \sum_{d \in R_w} x_i(d)$. Using linearity of expectation, we get $E[X_i] = |R_w|/2^i = N/2^i$. □

For $i = 0 \ldots M$, let γ_i denote the median of set T_i.

Lemma 2.3.2. *When asked for an estimate for the median, if* MedianQuery(w) *does not fail in Step 4, then it returns $\gamma_{\ell'}$ for value ℓ' selected in Step 1. Further, if $|R_w| \leq \alpha$, then* MedianQuery(w) *returns the exact median of R_w.*

Proof. Consider *MedianQuery(w)*. Note that the level chosen by the algorithm, ℓ', satisfies the condition that the timestamp of the most recently discarded element from ℓ' is less than $c - w$. Thus no element which has been selected into $S_{\ell'}$ and has a timestamp at least $c - w$ has been discarded. Next consider *MedianProcess(d)*. For any arriving element $d = (v,t) \in R_w$, if $x_{\ell'}(d) = 1$, there will be an insertion into $S_{\ell'}$ and by Definition 2.3.2, there will also be an insertion into T_i. If $x_{\ell'}(d) = 0$, the arrival of d will not cause an insertion into either $S_{\ell'}$ or into $T_{\ell'}$. Thus, the set of all elements in $S_{\ell'}$ that have timestamps at least $c - w$ is exactly the set $T_{\ell'}$. By returning the median of this set, the algorithm is returning $\gamma_{\ell'}$.

Suppose $|R_w| \leq \alpha$. Note that each element in R_w was selected into S_0 when it was processed. Since the α elements with the most recent timestamps are stored in S_0, it must be true that $R_w \subseteq S_0$. *MedianQuery* will retrieve all of R_w from S_0 and return the exact median of R_w. □

Because of the above lemma, in the rest of the proof, we assume that $|R_w| > \alpha$.

Definition 2.3.3. *For $i = 0 \ldots M$, let r_i denote the rank of γ_i in R_w. Define event B_i to be true if if $r_i \notin [(1/2 - \varepsilon)N, (1/2 + \varepsilon)N]$, and false otherwise. Define event G_i to be true if $(1 - \varepsilon)N/2^i \leq X_i \leq (1 + \varepsilon)N/2^i$, and false otherwise. Let $\ell^\star \geq 0$ be an integer such that $\alpha/4 < E[X_{\ell^\star}] \leq \alpha/2$.*

Lemma 2.3.3. *Level $\ell^\star$ is uniquely defined and exists for every input stream R.*

Proof. From Lemma 2.3.1, we have $E[X_i] = N/2^i$. Since $N > \alpha$, $E[X_0] > \alpha$. By the definition of $M = \lceil \log N_{max} \rceil$, it must be true that $N \leq 2^M$ for any input stream R, so that $E[X_M] \leq 1$. Since for every increment in i, $E[X_i]$ decreases by a factor of 2, there must be a unique level $0 < \ell^\star < M$ such that $E[X_{\ell^\star}] \leq \alpha/2$ and $E[X_{\ell^\star}] > \alpha/4$. □

The following lemma shows that for levels that are less than or equal to $\ell^\star$, the median of the random sample is very likely to be close (in rank) to the actual median of R_w. The proof uses conditional probabilities. We show that for levels that are less than or equal to ℓ^*, the number of elements selected into the the level is close to its expectation with high probability. Under this condition, we show the median of the sample is close to the actual median with high probability.

Lemma 2.3.4. *For* $0 \leq \ell \leq \ell^*$,

$$\Pr[B_\ell] < \frac{\delta}{2^{\ell^* - \ell + 2}}$$

Proof.

$$\begin{aligned} \Pr[B_\ell] &= \Pr[G_\ell \wedge B_\ell] + \Pr[\bar{G_\ell} \wedge B_\ell] \\ &\leq \Pr[B_\ell | G_\ell] \cdot \Pr[G_\ell] + \Pr[\bar{G_\ell}] \qquad (2.7) \\ &\leq \Pr[B_\ell | G_\ell] + \Pr[\bar{G_\ell}] \qquad (2.8) \end{aligned}$$

Using Lemmas 2.3.5 and 2.3.6 in Equation 2.8, we get:

$$\Pr[B_\ell] < \frac{5\delta/8}{2^{\ell^* - \ell + 2}} < \frac{\delta}{2^{\ell^* - \ell + 2}}$$

□

Lemma 2.3.5. *For* $0 \leq \ell \leq \ell^*$,

$$\Pr[\bar{G_\ell}] < \frac{\delta}{8 \cdot 2^{\ell^* - \ell + 2}}$$

Proof. Let $\mu_\ell = E[X_\ell] = N/2^\ell$

$$\begin{aligned}\Pr[\bar{G_\ell}] &= \Pr[X_\ell < (1-\varepsilon)\mu_\ell \vee X_\ell > (1+\varepsilon)\mu_\ell] \\ &\leq \Pr[X_\ell < (1-\varepsilon)\mu_\ell] + \Pr[X_\ell > (1+\varepsilon)\mu_\ell]\end{aligned}$$

Since $E[X_i] = N/2^i$ (from Lemma 2.3.1) and $E[X_{\ell^*}] > \alpha/4$, we have $\mu_\ell > (\alpha/4)2^{\ell^*-\ell}$. By Definition 2.1.2, we know $0 < \varepsilon < 1/2$. Using Lemma 2.2.6,

$$\begin{aligned}\Pr[\bar{G_\ell}] &\leq \Pr[X_\ell < (1-\varepsilon)\mu_\ell] + \Pr[X_\ell > (1+\varepsilon)\mu_\ell] \\ &\leq e^{-\mu_\ell\varepsilon^2/2} + e^{-\mu_\ell\varepsilon^2/3} \\ &\leq e^{(-\varepsilon^2\cdot\alpha\cdot 2^{\ell^*-\ell-3})} + e^{(-\varepsilon^2\cdot\alpha\cdot 2^{\ell^*-\ell-2}/3)} \\ &= (\frac{\delta}{8})^{2^{\ell^*-\ell+2}\cdot 3} + (\frac{\delta}{8})^{2^{\ell^*-\ell+3}} \\ &\leq 2(\frac{\delta}{8})^{2^{\ell^*-\ell+3}} \leq \frac{\delta/4}{2^{\ell^*-\ell+3}} = \frac{\delta}{2^{\ell^*-\ell+5}}\end{aligned}$$

We have used Lemma 2.2.7 in the last inequality. □

Lemma 2.3.6. *For* $0 \leq \ell \leq \ell^*$,

$$\Pr[B_\ell|G_\ell] < \frac{\delta}{2^{\ell^*-\ell+3}}$$

Proof.

$$\Pr[B_\ell|G_\ell] = \Pr[r_\ell < (1/2-\varepsilon)N|G_\ell] + Pr[r_\ell > (1/2+\varepsilon)N|G_\ell]$$

The proof will consist of two parts, Equations 2.9 and 2.10.

$$\Pr[r_\ell < (1/2-\varepsilon)N|G_\ell] < \frac{\delta/4}{2^{\ell^*-\ell+2}} \tag{2.9}$$

$$\Pr[r_\ell > (1/2+\varepsilon)N|G_\ell] < \frac{\delta/4}{2^{\ell^*-\ell+2}} \tag{2.10}$$

Proof of Equation 2.9: Let $L = \{d \in R_w | \text{rank of } d \text{ in } R_w \leq (1/2-\varepsilon)N\}$, $Y = \sum_{d\in L} x_\ell(d)$. By Lemma 2.3.1, we have $E[Y] = (1-2\varepsilon)N/2^{\ell+1}$ Since $r_\ell < (1/2-\varepsilon)N$, which means that at least the

smaller half elements in T_i were selected from the set L, combining the fact $X_i \geq (1-\varepsilon)N/2^{\ell}$, we have the following,

$$\begin{aligned}\Pr[r_\ell < (\frac{1}{2}-\varepsilon)N|G_\ell] &= \Pr[(r_\ell < (\frac{1}{2}-\varepsilon)N) \wedge G_\ell]/\Pr[G_\ell] \\ &\leq \Pr[Y \geq (1-\varepsilon)\frac{N}{2^{\ell+1}}]/\Pr[G_\ell] \\ &= \Pr[Y \geq (1+\delta')E[Y]]/\Pr[G_\ell],\end{aligned}$$

Where $\delta' = \varepsilon/(1-2\varepsilon)$ and $\Pr[G_\ell] \geq 1-\delta/(8\cdot 2^{\ell^*-\ell+2})$

Case 1: if $0 < \delta' < 1$, then

$$\Pr[Y \geq (1+\delta')E[Y]] \leq e^{-E[Y]\delta'^2/3} < (\frac{\delta}{8})^{\frac{2^{\ell^*-\ell+2}}{1-2\varepsilon}} < (\frac{\delta}{8})^{2^{\ell^*-\ell+2}} \leq \frac{\delta/8}{2^{\ell^*-\ell+2}}$$

Case 2: if $\delta' \geq 1$, then

$$\Pr[Y \geq (1+\delta')E[Y]] \leq e^{-E[Y]\delta'/3} < (\frac{\delta}{8})^{\frac{2^{\ell^*-\ell+2}}{\varepsilon}} < (\frac{\delta}{8})^{2^{\ell^*-\ell+2}} \leq \frac{\delta/8}{2^{\ell^*-\ell+2}}$$

Thus,

$$\Pr[Y \geq (1+\delta')E[Y]]/\Pr[G_\ell] \leq \frac{\delta}{2^{\ell^*-\ell+4}}$$

In both Cases 1 and 2, we have used the fact $E[X_\ell] = N/2^{\ell} > (\alpha/4)2^{\ell^*-\ell}$ in addition to Lemma 2.2.6 and Lemma 2.2.7. From Cases 1 and 2, Equation 2.9 is proved. Equation 2.10 can be similarly proved. From Equations 2.9 and 2.10, we get:

$$\Pr[B_\ell|G_\ell] < 2\frac{\delta}{2^{\ell^*-\ell+4}} = \frac{\delta}{2^{\ell^*-\ell+3}}$$

□

Lemma 2.3.7.

$$\sum_{i=0}^{\ell^*} \Pr[B_i] < \frac{\delta}{2}$$

Proof. The proof directly follows from Lemma 2.3.4

$$\sum_{i=0}^{\ell^*} \Pr[B_i] < \sum_{i=0}^{\ell^*} \frac{\delta}{2^{\ell^*-\ell+2}} = \delta \sum_{i=2}^{\ell^*+2} \frac{1}{2^i} < \delta \sum_{i=2}^{\infty} \frac{1}{2^i} = \frac{\delta}{2}$$

□

Recall that level ℓ' in *MedianQuery(w)* is used to answer the query for the median.

Lemma 2.3.8.

$$\Pr[\ell' > \ell^*] < \frac{\delta}{8}$$

Proof. If $\ell' > \ell^\star$, it follows that $|T_{\ell^\star}| = X_{\ell^\star} > \alpha$, else the algorithm would have stopped at a level less than or equal to $\ell^\star$. Thus, $\Pr[\ell' > \ell^\star] \leq \Pr[X_{\ell^\star} > \alpha]$. Let $Y = X_{\ell^\star}$. Since $Y = \sum_{d \in R_w} x_{\ell^*}(d)$, where $x_{\ell^*}(d)$ is 0-1 random variable, $E[Y] \leq \alpha/2$. Using Lemma 2.2.6, we have

$$\begin{aligned} \Pr[\ell' > \ell^\star] &\leq \Pr[Y > \alpha] \leq \Pr[Y > 2E[Y]] \\ &\leq e^{-E[Y]/3} < e^{-\alpha/12} < (\frac{\delta}{8})^{\frac{8}{\varepsilon^2}} \\ &< \frac{\delta}{8} \end{aligned}$$

We have used the fact $E[Y] > \alpha/4$. □

Theorem 2.3.1. *The result of algorithm* MedianQuery(w) *is an* (ε, δ)*-approximate median of* R_w.

Proof. Let f denote the probability that the algorithm fails to return an (ε, δ)-approximate median of R_w. Using Lemmas 2.3.7 and 2.3.8 and a similar argument to the proof of Theorem 2.2.1, we get:

$$\begin{aligned} f &= \Pr[\ell' > M] + \Pr[\bigcup_{i=0}^{M} (\ell' = i) \wedge B_i] \\ &\leq \Pr[\ell' > \ell^\star] + \sum_{i=0}^{\ell^\star} \Pr[B_i] < \left(\frac{\delta}{8} + \frac{\delta}{2}\right) \\ &< \delta \end{aligned}$$

□

2.3.3 Complexity

Lemma 2.3.9. Space Complexity: *The total space taken by the sketch for the median is* $O((1/\varepsilon^2)\log(1/\delta)\cdot\log N_{max}\cdot\sigma)$*, where* N_{max} *is an upper bound on the number of elements within* R_w*,* σ *is the space taken to store an input element* (v,t)*,* ε *is the desired relative error, and* δ *is the desired upper bound on the failure probability.*

Proof. The algorithm maintains $M = \lceil \log N_{max} \rceil$ samples, each of which has up to $\alpha = (96/\varepsilon^2)\ln(8/\delta)$ elements. Each element in the sample is a pair (v,t), which can be stored using σ bits. The product of the number of samples, the number of elements per sample, and the space per element yields the above space complexity. □

Lemma 2.3.10. Time Complexity: *The expected time taken for handling an element* (v,t) *is* $O(\log\log(1/\delta)+\log(1/\varepsilon))$*. The time taken to answer a query for the median is* $O(\log\log N_{max}+(1/\varepsilon^2)\log(1/\delta))$.

Proof. The proof is similar to that of Lemma 2.2.12. All elements in the same level can be stored in a heap. Each incoming element is sampled into an expected constant number of levels, where the cost of insertion into each level, plus the cost of handling the overflow is $O(\log\alpha)$. For answering a query for the median, the appropriate level can be found in time $O(\log\log N_{max})$ through a binary search, and finding the median of the sample at the appropriate level takes $O(\alpha)$ using the linear time algorithm for finding a median. So the total cost for answering a query for the median is $O(\log\log N_{max}+\alpha)$. □

2.4 Union of Sketches

In a distributed system, there could be multiple aggregators, each of which is observing a different local stream. It may be necessary to compute aggregates on not just any individual stream, but on the union of the data in all streams. We now consider the computation of aggregates over recent elements of the union of distributed data streams.

A simple solution to the above problem would be to send all streams directly to an aggregator (or the *sink*) which can then compute an aggregate on the entire data received. However, such an approach would be extremely resource-intensive with respect to communication complexity and energy, since each data item of each stream has to traverse a path from the source to the destination.

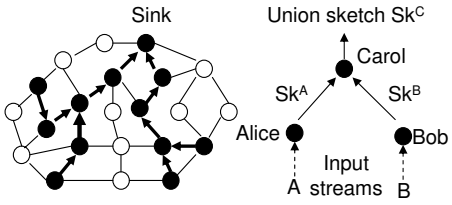


Figure 2.1 Left: a spanning tree which connects aggregators with flow of information towards a sink; Right: an aggregator merges the sketches of two aggregators

A much more efficient approach is for each node to compute a small space sketch of its local stream, and communicate the sketch to the sink. The sink can use the sketches to estimate the aggregate over the union of all data streams. Since the sketches are much smaller than the streams themselves, this approach has much smaller communication complexity than the simple approach. In sensor data processing, there have been successful proposals (for example, Madden *et al.* (56)) to combine such sketches in a hierarchical fashion, where sketches are combined up a spanning tree which is rooted at the sink node (see Figure 2.1).

Each aggregator sends its sketch to its parent. The parent node receives sketches from all its children, combines them into a new sketch and then sends the new sketch to its own parent. In this way, sketches propagate and get combined at intermediate levels of the tree until they reach the sink. The sink combines the received sketches from its children and produces a final sketch for the union of all the (local) streams received by all aggregators. We consider the simple case of three aggregators, Alice (child), Bob (child), and Carol (parent) (see Figure 2.1). The scenario can be generalized for an arbitrary number of aggregators, or aggregators organized in a hierarchy. Suppose Alice and Bob receive respective (asynchronous) streams A and B, producing sketches Sk^A and Sk^B, respectively, each for a maximum window size W. Alice and Bob transmit their sketches to Carol, who combines Sk^A and Sk^B to produce a sketch Sk^C for the union $A \cup B$. Though Carol never sees streams A or B, she can use Sk^C to answer aggregate queries for any timestamp window of width $w \leq W$ over the data set $A \cup B$.

The algorithm for the union is formally described in *Union*$(\cdot,\cdot)$. Given sketches Sk^A and Sk^B of streams A and B respectively, the algorithm outputs Sk^C, a sketch of $A \cup B$, which can be used to answer queries for the approximate sum or median of all elements within a sliding timestamp window over $A \cup B$. In *Union()*, for $0 \leq i \leq M$, let S_i^A denote the level i sample of Alice, and t_i^A de-

Algorithm 6: Union(Sk^A, Sk^B)

Input: $Sk^A =< S_0^A, t_0^A, S_1^A, t_1^A, , \ldots, S_M^A, t_M^A >$, a sketch of Alice's local stream A
$Sk^B =< S_0^B, t_0^B, S_1^B, t_1^B, \ldots, S_M^B, t_M^A >$, a sketch of Bob's local stream B

Output: Sk^C, a sketch of $C = A \cup B$

```
for i = 0...M do
    if |S_i^A ∪ S_i^B| ≤ α then
        S_i^C ← S_i^A ∪ S_i^B;
        t_i^C ← max{t_i^A, t_i^B};
    else
        S_i^C is the set of α most recent elements in S_i^A ∪ S_i^B;
        Let t be the most recent timestamp in ((S_i^A ∪ S_i^B) − S_i^C);
        t_i ← max{t, t_i^A, t_i^B};
```

note the most recent timestamp of a discarded element from S_i^A. The sketch computed by Alice is the vector $Sk^A = \langle S_0^A, t_0^A, S_1^A, t_1^A, \ldots, S_M^A, t_M^A \rangle$. Similarly, the sketch computed by Bob is the vector $Sk^B = \langle S_0^B, t_0^B, S_1^B, t_1^B, \ldots, S_M^B, t_M^B \rangle$.

The high level algorithm for the union is the same whether the aggregate required is the sum or the median. The only difference is that in case of the sum, the parameter $M = \log V_{max}$, where V_{max} is an upper bound on the sum of observations within the window across all streams. Note that the sketch of each local stream must also use $M = \log V_{max}$, where V_{max} is defined above. In case of the median, $M = \log N_{max}$, where N_{max} is an upper bound on the number of elements within the timestamp window across all streams. Note that the sketch of each local stream must also use $M = \log N_{max}$, where N_{max} is defined above. Of course, the algorithms for sketching the local streams are different for the sum and for the median, though the algorithms for the union of sketches are the same. The initialization of Sk^C and the algorithm for answering the query (sum or median) using Sk^C are the same as for the single stream case.

The sketches can be combined hierarchically. For example, suppose D and E were two other local streams, and Sk^F was the result of *Union*(Sk^D, Sk^E). Then Sk^C and Sk^F can be combined using *Union()* to yield a sketch of $A \cup B \cup D \cup E$. A key property required for the above hierarchical union to work is that the combination of sketches is *lossless* and *compact*. A sketch is said to be *compact* if Sk^C, the sketch resulting from the Union operation, has the same (small) upper bound on the size as do Sk^A and Sk^B. If a sketch is compact, then the size of the sketch resulting from the combination of many sketches

is bounded, and does not increase beyond a threshold no matter how many sketches are combined. The sketch is said to be *lossless* if the guarantee provided by Sk^C (for example, (ε,δ)-accuracy for the sum or the median) on data $A \cup B$ is the same as the guarantees provided by sketches Sk^A and Sk^B on data sets A and B respectively. In this way, the quality and size of the sketch at each level of the tree will be insensitive to the structural properties of the tree, such as its degree and depth. We now argue that the sketches developed for the sum and the median are compact.

Compactness of Sketches. The sketch resulting from the union Sk^C is $< S_0^C, t_0^C, S_1^C, t_1^C, \ldots, S_M^C, t_M^C >$. Since the upper bound on the number of elements in S_i^C, S_i^B and S_i^A are all α, the bounds on the sizes of Sk^C, Sk^A and Sk^B are identical. Thus, the sketch for the sum is compact and has a space complexity as described in Lemma 2.2.11. Similarly, the sketch for the median is compact, and has a space complexity as described in Lemma 2.3.9.

Losslessness of Sketches. We will now show that the sketch resulting from *Union()* also preserves the same accuracy as its constituent sketches, but for the data stream constructed by the union of the individual streams. Let Sk_{sum}^A and Sk_{sum}^B respectively denote the sketches for A and B for the sum, for a maximum window size W, relative error ε and failure probability δ. Let Sk_{med}^A and Sk_{med}^B respectively denote the sketches for A and B for the (ε,δ)-approximate median, for a maximum window size $W \geq w$. Let Sk_{sum}^C denote the result of $Union(Sk_{sum}^A, Sk_{sum}^B)$ and Sk_{med}^C denote the result of $Union(Sk_{med}^A, Sk_{med}^B)$.

Let $Sk_{sum}^{A\cup B}$ be the sketch resulting from applying Algorithm *SumProcess* (described in Section 2.2) for over all elements of the stream $A \cup B$. In generating the sketch of $A \cup B$, we do not assume anything about the order of arrival of elements in $A \cup B$; the resulting sketch will not depend on this order. We assume that the random choices that are made by algorithm *SumProcess* in processing an element are identical whether the element is processed as a part of stream $A \cup B$ or as a part of the individual streams A or B. The sketch for the sum assumes a maximum window size W, relative error ε and failure probability δ. Similarly, let $Sk_{med}^{A\cup B}$ be the sketch resulting by applying algorithm *MedianProcess* (described in Section 2.3) over all elements of $A \cup B$. Again, the elements of stream $A \cup B$ can arrive in any order, and this order does not affect the final sketch generated. The sketch for the median also assumes a maximum window size W, and returns an (ε,δ)-approximate median. For simplicity,

we assume that a sketch never contains any element with a timestamp of less than $(c-W)$, where c is the current time. This assumption is justified since the algorithms *SumQuery* (Section 2.2) and *MedianQuery* (Section 2.3) will never consider such elements with timestamps less than $(c-W)$, even if they are present in the sketch.

Lemma 2.4.1.

$$Sk^C_{sum} = Sk^{A\cup B}_{sum}$$

$$Sk^C_{med} = Sk^{A\cup B}_{med}$$

Proof. We show $Sk^C_{sum} = Sk^{A\cup B}_{sum}$, and a similar argument holds for $Sk^C_{med} = Sk^{A\cup B}_{med}$. Let R^A_W and R^B_W denote the set of elements with timestamps in the maximum window $[c-W,c]$ over streams A and B respectively. For $i = 0,1,\ldots,M$, let $S^A_{sum,i}$ denote the ith level sample of Sk^A_{sum}. Similarly, we define $S^B_{sum,i}, S^C_{sum,i}$ and $S^{A\cup B}_{sum,i}$.

For any element $(v,t) \in R^A_W \cup R^B_W$, note that the same algorithm *SumProcess* is used to process the element, whether it occurs as an element in stream A or B or $A\cup B$. *SumProcess* uses only v and t to decide whether the element (v,t) is selected into the sample at level i or not. Thus, if $(v,t) \in R^A_W$ is selected into $S^A_{sum,i}$, then it will also be selected into $S^{A\cup B}_{sum,i}$, and vice versa. Therefore the set of elements that are ever selected into $S^A_{sum,i}$ or $S^B_{sum,i}$ is exactly the same set of elements that are ever selected into $S^{A\cup B}_{sum,i}$. For any level $i = 0\ldots M$, $S^A_{sum,i}$ retains the α elements with the most recent timestamps that were ever selected into $S^A_{sum,i}$, and similarly with $S^B_{sum,i}$. From Steps 2 and 5 of *Union()*, we see that $S^C_{sum,i}$ retains the α most recent elements that ever selected into $S^A_{sum,i}$ or into $S^B_{sum,i}$. Thus, $S^C_{sum,i}$ retains the α most recent elements among $A\cup B$ that were selected into level i of Sk^A or Sk^B.

Note that $S^{A\cup B}_{sum,i}$ also keeps the α most recent elements that are ever selected into $S^{A\cup B}_{sum,i}$. Thus, for $i = 0,1,\ldots,M$, $S^C_{sum,i} = S^{A\cup B}_{sum,i}$, which implies $Sk^C_{sum} = Sk^{A\cup B}_{sum}$. □

From the above lemma, it follows that all properties of $Sk^{A\cup B}_{sum}$ carry over to Sk^C_{sum}. From Theorem 2.2.1 we know $Sk^{A\cup B}_{sum}$ provides an (ε,δ)-estimate for the sum within any timestamp window of width at most W on $A\cup B$. Thus, Sk^C_{sum} also provides the same estimate for the sum within a sliding window, showing that the union of sketches is lossless. A similar argument can be made for sketches

for the median.

We now consider sketches that are combined in a hierarchical fashion. Consider a tree where each leaf observes a local stream, and passes a sketch for the sum (median) up to its parent. Sketches arriving at any internal node are combined and passed up the tree until the root receives sketches from all its children. If the algorithm *Union()* was applied at every internal node of the tree, then the root will finally have a sketch that can be used to answer queries for the sum (median) of elements within a sliding timestamp window of the union of all streams appearing at the leaves of the tree. This can be proved by repeatedly applying Lemma 2.4.1 at every internal node of the tree and at the root. The above algorithm for the union applies even if the intermediate nodes of the hierarchy had local streams.

2.5 Concluding Remarks

In this chapter, we presented algorithms for sketching asynchronous data streams over a sliding window of the most recent elements. Our sketches are based on random sampling and can return the approximate sum or the approximate median of elements within the sliding window. We note that the same technique that was used for the median can also be used to maintain approximate quantiles of elements within the sliding window. These sketches are also useful in distributed computations since they can be composed in a compact and lossless manner.

CHAPTER 3. General Time-decay Based Processing

This chapter presents a new sketch for summarizing general purpose network streaming data. It is an generalization of the sketch in Chapter 2. The new sketch has the following properties that make it useful in communication-efficient aggregation in distributed streaming scenarios, such as sensor networks: (1) The sketch can handle asynchronous data streams. (2) The sketch is duplicate-insensitive, i.e. reinsertions of the same data will not affect the sketch, and hence the estimates of aggregates. (3) Unlike previous duplicate-insensitive sketches for sensor data aggregation (64; 22), it is also time-decaying, so that the weight of a data item in the sketch can decrease with time according to any arbitrary user-specified decay function, including the sliding window. (4) The sketch can give provably approximate guarantees for various aggregates of data, including the sum, median, quantiles, and frequent elements. (5) The size of the sketch and the time taken to update it are both polylogarithmic in the size of the relevant data. (6) Further, multiple sketches computed over distributed data streams can be combined without loss of accuracy. To our knowledge, this is the first sketch that combines all the above properties.

3.1 Introduction

We motivate the design of this new sketch for communication efficient data aggregation in distributed data stream scenarios by looking at its usage in the wireless sensor networks as an example. 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 (56). 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 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) (64; 22). 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 sliding windows, and ignore (assign zero weight to) any that are older, as we do in Chapter 2; or, more generally, use an arbitrary decay function that assigns a weight to each observation (21). 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.

3.1.1 Problem Formulation

Data Stream. We use the asynchronous data stream defined in Section 1.3 to model the stream of observations seen by a single sensor: $R = \langle e_1, e_2, \ldots, e_n \rangle$, where each e_i is a tuple (v_i, w_i, t_i, id_i). Recall that it is possible that same observations appear multiple times in the stream, but only one copy of the duplicates should be counted in computing the aggregates. As we have explained in Section 1.3 and will see more examples in this chapter, the abstraction of this data stream model captures a wide variety of cases that can be encoded in this form.

Decay Functions. Recall that a decay function $f(w,x)$ takes the initial weight w and the age x of the stream element (v,w,t,id), and returns the decayed weight of the element at any clock time c. The age of the element at time c is defined as $c-t$, the elapsed time since the element was created. (Section 1.4)

Definition 3.1.1. *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: $f(w,x) = \lfloor w/2^x \rfloor$.

3.1.2 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 v such that the total decayed weight of all elements in D whose value is less than or equal to v 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 (61), 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 $0 < \varepsilon < \phi$, the ε-approximate decayed ϕ-quantile is a value v such that the relative rank of v is at least $\phi - \varepsilon$ and the relative rank of $v-1$ is less than $\phi + \varepsilon$.

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 v such that $\psi(v) > \phi$ 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 (7). Thus we consider the ε-approximate

frequent elements problem, which requires us to return all values v such that $\psi(v) > \phi$ and no value v' such that $\psi(v') < \phi - \varepsilon$.

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 $|\hat{Q} - Q| \leq \varepsilon$.

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 $\delta < 1/2$ (5), and for a sketch that can give a deterministic approximation (5; 53); 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".

3.1.3 Contribution

The main contribution of this work is a general purpose sketch that can estimate all the above aggregates in a general model of network 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 (Definition 1.4.2). 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 $1/\varepsilon$, where ε is the relative error. There are lower bounds (49) showing that the quadratic dependence on $1/\varepsilon$ is necessary for duplicate insensitive computations on data streams, thus implying that our upper bounds are close to optimal.

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.

Our algorithm for an integral decay function is based on random sampling, and this chapter 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).

Outline of this chapter. After describing related work in Section 3.2, we consider the construction of a sketch for the case of integral decay in Section 3.3. Although such functions initially seem limiting, they turn out to be the key to solving the class of decomposable decay functions efficiently. In Section 3.4, 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 3.5, we present the results of our experiments. We make some concluding observations in Section 3.6.

3.2 Related Work

There is a large body of work on data aggregation algorithms in the areas of data stream processing (63) and sensor networks (51; 3; 19). 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 (39) 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 (64) 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 (13) to draw a uniform, unweighted sample of observed items. Also leveraging the FM sketch (39), Considine, Li, Kollios and Byers (22) 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 (27) show duplicate insensitive computations of quantiles, heavy hitters, and frequency moments. They do not consider the time dimension either.

Datar, Gionis, Indyk and Motwani (35) 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 (42) gave an algorithm for basic counting based on a data structure called *wave* with improved worst-case performance. Subsequently, Braverman and Ostrovsky (12) 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 (7) 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 (54). Babcock, Datar, Motwani and O'Callaghan (10) 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 (35), Section 7.5, where an approach to count the distinct values in a sliding window is briefly described.

Cohen and Strauss (21) 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 chapter, 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 (21)): performing an approximate duplicate-insensitive count (even without time decay) requires randomization in order to achieve sublinear space (5). Subsequently, Kopelowitz and Porat (52) 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 (20) considered spatially-decaying aggregation over network data, based on tracking lists of identities 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 (9) 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 (2) 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 (2) 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 (41) 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 (41), 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 (67) to compute the duplicate-insensitive sum as well as other aggregates such as max-dominance norm. Xu, Tirthapura, and Busch (79) 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.

3.3 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.

3.3.1 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 3.3.1.

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

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 3.3.4. 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 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 expiry(e, i). For any queries issued at time $\kappa \geq$ 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 3.3.3, we present a fast algorithm to compute 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 (67), 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 expiry$(e_4, 1) > 15$, expiry$(e_5, 1) > 15$, expiry$(e_6, 1) > 15$ and expiry$(e_2, 1) \leq 15$),

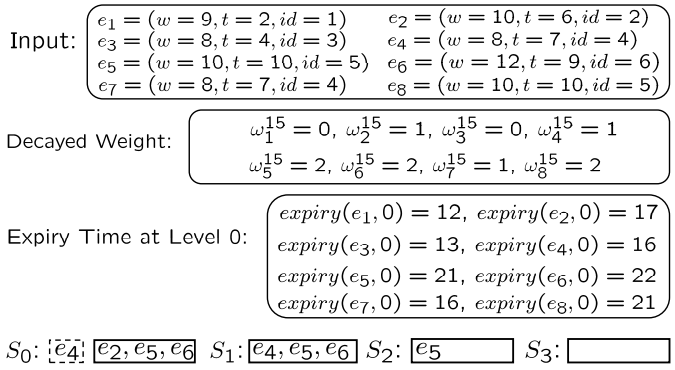


Figure 3.1 An example stream with 8 elements arriving in the order $e_1, e_2, \ldots, e_8$, and its sketch $\{S_0, S_1, S_2, S_3\}$ for the decayed sum. The current time is 15. The decayed weight of e_i at time t is denoted by ω_i^t. The expiry time of e_i at level j is denoted by expiry(e_i, j). The 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.

and since there is enough room, all these are stored in S_1. Only e_5 is selected into S_2 and no element is selected into level 3.

When a query is posed for the sum at time 15, the algorithm finds the smallest number ℓ such that the sample S_ℓ has not discarded any element whose expiry time is greater than 15. For example, in Figure 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.

3.3.2 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 (16)). 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

Algorithm 7: Initialization(M)

```
1 Randomly choose a hash function h as described in Section 3.3.2;
2 for 0 ≤ i ≤ M do
3     S_i ← ∅;
4     t_i ← −1 ;    /* t_i is maximum expiry time of all the elements discarded so
                       far at level i */
```

time of an element $e = (v, w, t, id)$ at sample level i as follows.

Let $A_e^i = \{\bar{t} \geq t : |r_e^{\bar{t}}| > 0 \text{ and for all } x \in r_e^{\bar{t}}, h(x) > \lfloor 2^{-i} p - 1 \rfloor\}$. 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 $B_e = \{\bar{t} \geq t : |r_e^{\bar{t}}| = 0\}$. 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 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 3.3.1. *For stream element $e = (v, w, t, id)$, and level $0 \leq i \leq M$:*

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

Intuitively, $\text{expiry}(e, i)$ 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 7 and 8.

Lemma 3.3.1. *The sample S_i is order insensitive; it is unaffected by permuting the order of arrival of the stream elements. The sample is also duplicate insensitive; if the same element e is observed*

Algorithm 8: ProcessItem(e=(v, w, t, id))

```
1 for 0 ≤ i ≤ M do
2     if (e ∈ S_i) then return ;                          /* e is a duplicate.  */
3     if (expiry(e,i) > max{c,t_i}) then
4         S_i ← S_i ∪ {e};
5         if |S_i| > τ then                                /* overflow */
6             t_i ← min_{e∈S_i} expiry(e,i);
7             S_i ← S_i \ {e : expiry(e,i) = t_i};
```

Algorithm 9: MergeSketches(S, S')

```
1 for 0 ≤ i ≤ M do
2     S_i ← S_i ∪ S'_i ;
3     t_i ← max{t_i, t'_i};
4     while |S_i| > τ do
5         t_i ← min_{e∈S_i} expiry(e,i) ;
6         S_i ← S_i \ {e : expiry(e,i) = t_i};
```

multiple times, the resulting sample is the same as if it had been observed only once.

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

Theorem 3.3.1. *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 9. □

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 3.3.2 (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 3.3.3). 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. □

3.3.3 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,

Algorithm 10: ExpiryTime(e,i)

Input: $e=(v,w,t,id)$, i, $0 \le i \le M$
Output: $expiry(e,i)$

```
1  Δ_e^i ← MinHit(p, a, h(w_max · id), w · f(0) − 1, ⌊2^{−i} p⌋ − 1) ;   /* h(x) = (ax+b) mod p */
2  if Δ_e^i ≥ 0 then                                                    /* s_e^i exists */
3      l ← 0;
4      r ← t_max ;
5      t' ← ⌊(l + r)/2⌋;
6      while t' ≠ l do                                                  /* Binary Search for t' */
7          if (f(w, t') > Δ_e^i) then l ← t';
8          else r ← t';
9          t' ← ⌊(l + r)/2⌋;
10     return t + t';
11 else return t ;                                                      /* s_e^i does not exist */
```

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 3.3.2. *If* $\Delta_e^i \ge 0$, *then* expiry$(e,i) = t + t'$, *where* $t' = \min\{\overline{t} : f(w,\overline{t}) \le \Delta_e^i\}$. *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 10.

Algorithm 11: 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                                   */
if (p ≤ L or u ≤ L) then return 0;
else if (a = 0) then return −1;
else
    Compute |S_0| ;        /* S = {u, (u+a) mod p, ··· , (u+n·a) mod p} = S_0 S_1 ··· S_k   */
    if (|S_0| = n+1) then return −1;
    else if (a = 1) then return (p−u);
/* Recursive Calls                                                  */
r ← p mod a ;
Compute k, f_1 ;                          /* f_1 is the first element of S_1   */
if (a − r ≤ a/2) then d ← MinHit(a, a−r, f_1, k−1, L) ;
else d ← MinHit(a, r, (a − f_1 + L) mod a, k−1, L) ;
/* Recursive Call Returns                                           */
if (d ≠ −1) then
    Compute f_{d+1} ;
    d ← [(d+1)p − u + f_{d+1}]/a ;
return d;
```

We can now focus on the efficient computation of Δ_e^i. One possible solution, presented in a preliminary version of this work (30), 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 (67) 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 11. 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 $h(l),h(l+1),\ldots,h(r)$ 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 $P=\{j:0\le j\le n,(u+j\cdot a) \bmod p\le L\}\neq\emptyset$, then $d=\min(P)$; else, $d=-1$.*

Let S denote the following arithmetic progression on Z_p: $\langle u \bmod p,(u+a) \bmod p,\ldots,(u+n\cdot a)$

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 (67), 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 3.3.3. *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 $f_m = S[d']$. 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 3.3.3, we can obtain d directly.

Lemma 3.3.4. *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 $f_0 = u \mod p$ is known. □

The next observation from (67) is crucial for finding m.

Observation 3.3.2 (Observation 2, Section 3.1 (67))**.** *Sequence $\bar{F} = F \setminus \{f_0\}$ is an arithmetic progression over Z_a, with common difference $a - r$ (or $-r$ equivalently), where $r = p \mod a$.*

So, we have two possibilities: (1) If $f_0 \leq L$, 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:

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

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 (67), 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:

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

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 (67), however, it is not clear how to further apply the technique in (67) 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:

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

We note that the idea of shifting the origin of the ring is very simple and useful. Using this idea simplifies the *Hits* algorithm in (67) since all the additional operations dealing with the effect of flipping the ring can be omitted.

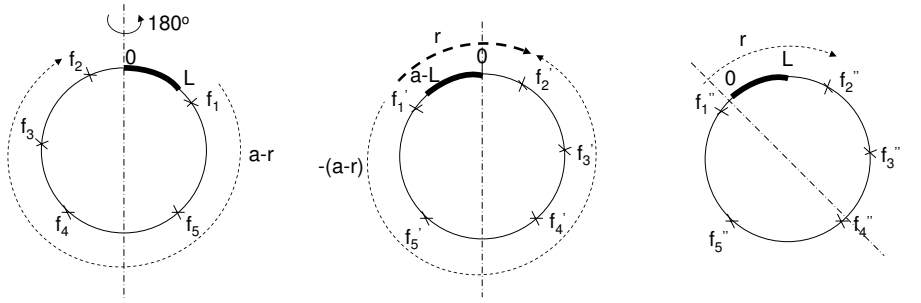
The above visualized intuition in case 2 is validated by the following lemma.

Lemma 3.3.5. *Given p, a, u, n, L as in Problem 1, set $P = \{i : 0 \leq i \leq n, (u + i \cdot a) \mod p \leq L\}$ and $P' = \{j : 0 \leq j \leq n, ((p - u + L) \mod p + j \cdot (p - a)) \mod p \leq L\}$, then:*

$$P = P'$$

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

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



(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 $\bar{F''}$

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

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

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

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

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 $(-\sigma) \mod p = p-\sigma \leq L$, i.e., $L+\sigma \geq P$, this yields a contradiction. Therefore, $(u+\gamma \cdot a) \mod p \leq L$. 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 3.3.6. *The algorithm* MinHit(p,a,u,n,L) *(shown in Algorithm 11) 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 $d = \min\{i : 0 \leq i \leq n, (u+i \cdot a) \mod p \leq L\}$, if such d exists; otherwise, return $d = -1$. Clearly, if $p \leq L$ or $u \leq L$, $d = 0$. 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 $f_1 = S[p-u] = 0 \leq L$, 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 3.3.3, if $d \neq -1$, we know $S[d] \in \bar{F}$. Because of Observation 3.3.2, we can make a recursive call at lines 9 or 10, to find j, $1 \leq j \leq k$, such that $f_j = S[d]$. Because of Lemma 3.3.5, lines 9 and 10 return the same result (with different time cost though). Using the formula presented in Lemma 3.3.4, 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 $n_{new} \leq \lceil n \cdot a/p \rceil$ and $a \leq p/2$ always hold in every recursive call, thus we have $n_{new} \leq n/2$, 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 $L \geq p$, 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 (67), 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 to $O(\log p + \log n)$. □

Theorem 3.3.3. *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 3.3.2, given Δ_e^i algorithm ExpiryTime() computes expiry(e,i) using additional $O(\log t_{max})$ time for the binary search. □

Algorithm 12: DecayedSumQuery(c)

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

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 $w' \le w$, $\min\{x : f(w,x) = w'\}$ 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 $f(w,x) = \lfloor w/a^x \rfloor$, given $\Delta_e^i \ge 0$ (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 $\text{expiry}(e,i) = \left\lfloor \log_a(w/(\Delta_e^i+1)) \right\rfloor + t + 1$, 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 $\Delta_e^i \ge 0$, then $\text{expiry}(e,i) = t + W$, where $e = (v,w,t,id)$ and W is the window size.

3.3.4 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 $p_i = \lfloor p2^{-i} \rfloor / p$ 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 $\tau = 60/\varepsilon^2$, and the maximum number of levels be $M = \lceil \log w_{max} + \log id_{max} \rceil$.

Theorem 3.3.4. *For any integral decay function f, Algorithm 12 yields an estimator $\hat{V}$ of V such that* $\Pr[|\hat{V} - V| < \varepsilon V] > 2/3$. *The time taken to answer a query for the sum is* $O(\log M + (1/\varepsilon^2)\log w_{max})$. *The expected time for each update is* $O(\log w(\log(1/\varepsilon) + \log w + \log t_{max}))$. *The space complexity is* $O((1/\varepsilon^2)(\log w_{max} + \log id_{max}))$.

Proof. We show the correctness of our algorithm for the sum through a reduction to the range-efficient algorithm for counting distinct elements from (67) (we refer to this algorithm as the PT algorithm, for the initials of the authors of (67)). Suppose a query for the sum was posed at time c. Consider the stream $\mathscr{I} = \{r_e^c : e \in R\}$, which is defined on the weights of the different stream elements when the query is posed. From Observation 3.3.1, we have $|\cup_{r\in\mathscr{I}} r| = V$.

Consider the processing of the stream $\mathscr{I}$ by the PT algorithm. The algorithm samples the ranges in $\mathscr{I}$ into different levels using hash function h. When asked for an estimate of the size of $\cup_{r\in\mathscr{I}} r$, 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 $Y = (1/p_{\ell'}) \sum_{e\in D} \text{RangeSample}(r_e^c, \ell')$. From Theorem 1 in (67), Y satisfies the condition $\Pr[|Y - V| < \varepsilon V] > 2/3$ if we choose the sample size $\tau = 60/\varepsilon^2$, 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 12 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 3.3.2.

□

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 13. We show that by setting $\tau = 492/\varepsilon^2$ and $M = \lceil \log w_{max} + \log id_{max} \rceil$, we can get Theorem 3.3.5.

The following process only helps visualize the proof, and is not executed by the algorithm. Since the sketch is duplicate insensitive (Lemma 3.3.1), 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 $Q = V'/V$, for which we compute an estimate Q'. Recall that the sample size $\tau = C/\varepsilon^2$, where C is a constant to be determined through the analysis.

The next part of this section, from Fact 3.3.1 through Lemma 3.3.15, helps in the proof of Theorem 3.3.5 (stated formally below). The proof idea is similar to the one for Theorem 2.3.1.

Fact 3.3.1 (Fact 1 in (67))**.** *For any* $i \in [0 \ldots M]$, $1/2^{i+1} \leq p_i \leq 1/2^i$

Lemma 3.3.7. *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 3.3.2. *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 3.3.3. *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 3.3.4. *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 3.3.8. *For any $e \in r_d^c, d \in D$, $\mathsf{E}[x_i(e)] = p_i$, $\sigma^2_{x_i(e)} = p_i(1-p_i)$, $0 \le i \le M$.*

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

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

Lemma 3.3.9. *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 3.3.5. *For $i = 0, 1, \ldots, M$, define event B_i to be true if $Q' \notin [Q-\varepsilon, Q+\varepsilon]$, and false otherwise; define event G_i to be true if $(1-\varepsilon/2)p_i V \le X_i \le (1+\varepsilon/2)p_i V$, false otherwise.*

Definition 3.3.6. *Let $\ell^* \ge 0$ be an integer such that $\mathsf{E}[X_{\ell^*}] \le \tau/2$ and $\mathsf{E}[X_{\ell^*}] > \tau/4$.*

Lemma 3.3.10. *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 $\mathsf{E}[X_M] \le 1$. 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 $\mathsf{E}[X_{\ell^*}] \le \alpha/2$ and $\mathsf{E}[X_{\ell^*}] \ge \alpha/4$. □

From now on we consider the case with $0 < Q \le 1/2$. By symmetry, a similar proof exists for the case with $1/2 \le Q < 1$. 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 3.3.11. *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 3.3.12 and 3.3.13 in Equation 3.1, we get:

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

□

Lemma 3.3.12. *For* $0 \leq \ell \leq \ell^*$,

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

Proof. By Lemma 3.3.9, $\mu_{X_\ell} = p_\ell V, \sigma^2_{X_\ell} = p_\ell(1-p_\ell)V$, and by Chebyshev's inequality, we have

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

The last inequality is due to the fact: $\mu_{X_\ell} \geq 2^{\ell^* - \ell} \cdot \mu_{X^*_\ell} > 2^{\ell^* - \ell} \cdot \tau/4 = 2^{\ell^* - \ell - 2} \cdot C/\varepsilon^2$, using Fact 3.3.1

□

Lemma 3.3.13. *For* $0 \leq \ell \leq \ell^*$,

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

Proof.

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

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

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

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

Proof of Equation 3.2: Let $Y = \sum_{y\in I'} x_\ell(y) = Q'X_\ell > (Q+\varepsilon)X_\ell$. By Lemma 3.3.9, we have $\mu_Y = p_\ell VQ$, $\sigma_Y^2 = p_\ell(1-p_\ell)VQ$. Using Chebyshev's inequality and the fact $X_\ell \geq (1-\varepsilon/2)p_\ell V$, we have the following,

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

The last three inequalities use the facts: $(1-p_\ell)Q<1$, $(Q+\varepsilon)(1-\varepsilon/2)\geq Q+\varepsilon/2$ due to $0<\varepsilon<Q\leq 1/2$, $p_\ell V > 2^{\ell^*-\ell}\tau/4$ 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 3.3.14.

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

Proof. The proof directly follows from Lemma 3.3.11.

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

□

Lemma 3.3.15.

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

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, $\Pr[\ell > \ell^\star] \leq \Pr[X_{\ell^\star} > \tau]$. Let $Y = X_{\ell^\star}$. By Lemma 3.3.9, Chebyshev's inequality and the fact $\mu_Y < \tau/2$, we have

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

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

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

□

Theorem 3.3.5. *For any integral decay function f, Algorithm 13 yields an estimate $\hat{Q}$ of Q such that $\Pr[|\hat{Q} - Q| < \varepsilon] > 2/3$. The time taken to answer a query for the selectivity of P is $O(\log M + (1/\varepsilon^2)\log w_{max})$. The expected time for each update is $O(\log w(\log(1/\varepsilon) + \log w + \log t_{max}))$. The space complexity is $O((1/\varepsilon^2)(\log w_{max} + \log id_{max}))$.*

Proof. Let f denote the probability that the algorithm fails to return an ε-approximate selectivity estimation of D. By using Lemmas 3.3.14 and 3.3.15 and $C = 492$, we get:

$$f = \Pr[\ell > M] + \Pr[\bigcup_{i=0}^{M} (\ell = i) \wedge B_i] \leq \Pr[\ell > \ell^\star] + \sum_{i=0}^{\ell^\star} \Pr[B_i] < \frac{164}{C} = \frac{1}{3}$$

The query time complexity analysis is similar to the one for the sum in Theorem 3.3.4. The expected time for each update and the space complexity directly follows from Theorem 3.3.2. □

Algorithm 13: DecayedSelectivityQuery(P,c)

$\ell = \min\{i : 0 \le i \le M, t_i \le c\}$;

if ℓ *does not exist* **then return** ; `/* the algorithm fails */`

if ℓ *exists* **then return** $\frac{\sum_{e=(v,w,t,id)\in S_\ell} RangeSample(r_e^c,\ell)\cdot P(v,w)}{\sum_{e\in S_\ell} RangeSample(r_e^c,\ell)}$;

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 3.3.6. *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 $O(\log M + (1/\varepsilon^2)\log(w_{max}/\varepsilon))$. The expected time for each update is $O(\log w(\log(1/\varepsilon) + \log w + \log t_{max}))$.The space complexity is $O((1/\varepsilon^2)(\log w_{max} + \log id_{max}))$.*

Proof. The expected time for each update and the space complexity directly follows from Theorem 3.3.2. Now we show how to reduce a sequence of problems to 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 v asks to estimate the (weighted) fraction of elements whose value v is at most v. This is encoded by a predicate $P_{\le v}$ such that $P_{\le v}(v,w) = 1$ if $v \le v$, 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 v, regarding the predicate "$P_{=v}(v,w) = 1$ if $v = v$", the selectivity of v, which is also the frequency of v, in the stream is ϕ or more with additive error at most ε with probability at least $1 - \delta$.

□

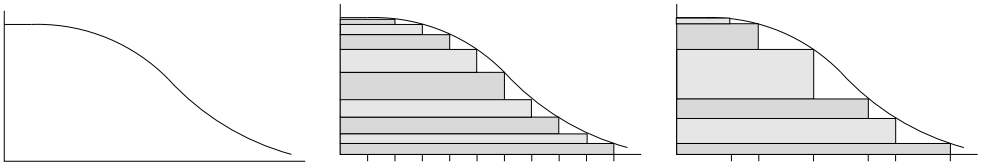
3.4 Decomposable Decay Functions via Sliding Window

3.4.1 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 (Section 1.4.1). 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 3.3. We can compute the expiry time of any element e at level ℓ in $\log w$ time as $(t+W)$ if $\Delta_e^\ell \geq 0$; t, otherwise. We can prove a stronger result though: If we set $f(w,x) = w$ for all $x \geq 0$ 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 3.4.1. *Our data structure can answer sliding window sum and selectivity queries where the parameter W is provided at query time. Precisely, for $\tau = O(1/\varepsilon^2)$ 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 $\Pr[|\hat{V} - V| < \varepsilon V] > \frac{2}{3}$ and an estimate $\hat{Q}$ of the sliding window decayed selectivity, Q, such that $\Pr[|\hat{Q} - Q| < \varepsilon] > \frac{2}{3}$. 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



(a) (b) (c)

Figure 3.3 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.

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 3.3.4 and 3.3.5.

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.

3.4.2 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 (21):

rewrite the decayed computation as the combination of many sliding window queries, over different sized windows. We further show how this reduction can be done in a time-efficient manner.

Selectivity Estimation

Lemma 3.4.1. *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

$$Q = \sum_{(v,w,t,id)\in D} w \cdot P(v,w) \cdot g(c-t) / \sum_{(v,w,t,id)\in D} w \cdot g(c-t), \tag{3.4}$$

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 $(1 \pm \varepsilon)$ 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 $(1 \pm \varepsilon)$ 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 3.4.1 implies each sliding window query in A is accurate up to a $(\pm\varepsilon V)$ additive error with probability at least $1 - \delta$. This analysis further yields an estimate for A with the accuracy up to $(\pm\varepsilon V)$ additive error with probability at least $1 - c\delta$. Combining the estimates for A and B and using

$\tau = O(1/\varepsilon^2)$, we get $|Q' - Q| \leq \varepsilon$ 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 $O(\log c)$. □

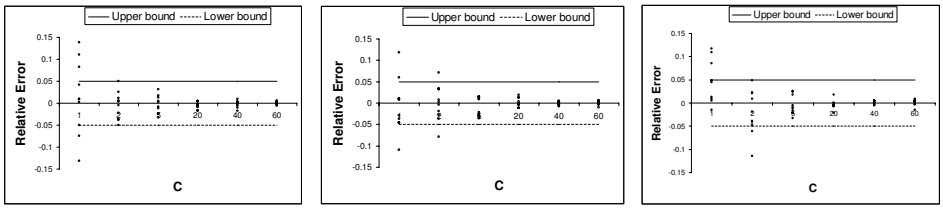
Theorem 3.4.2. *We can answer a selectivity query using an arbitrary decomposable decay function* $f(w,x) = w \cdot g(x)$ *in time* $O(M\tau \log(\frac{M\tau}{\delta}) \log(M\tau \log \frac{M\tau}{\delta}))$ *to find* $\hat{Q}$ *so that* $\Pr[|Q - \hat{Q}| > \varepsilon] < \delta$.

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

$$\sum_{i=0}^{M} \sum_{\substack{j=1 \\ t_i^j < t_{i-1}^{min}}}^{|S_i|} (g(c - t_i^j) - g(c - t_i^{j+1}))V^{c-t_i^j} \tag{3.5}$$

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 3.3: 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



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

Figure 3.4 Decayed Sum: Accuracy vs C ($\varepsilon = 0.05$)

each step, which can be maintained by standard dictionary data structures in time $O(\log b)$. If we exhaust a level in any structure, then we move to the next level and find the appropriate place based on the current timestamp. In this way, we work through each data structure in a single linear pass, and spend time $O(\log b)$ for every time step we pass. Overall, we have to collect and sort $O(M\tau b)$ timestamps, and perform $O(M\tau b)$ probes, so the total time required is bounded by $O(M\tau b \log(M\tau b))$. This yields the bounds stated above. □

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

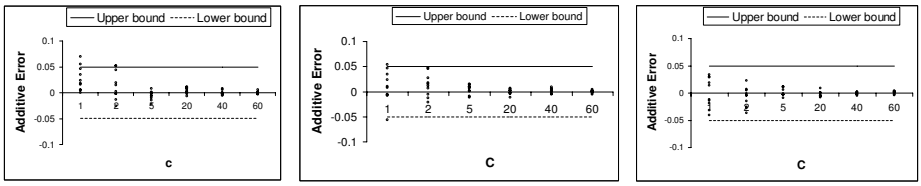
Decayed Sum Computation We observe that the maintenance of the decayed sum over general decay functions has already been handled as a subproblem within selectivity estimation.

Lemma 3.4.2. *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 3.4.3. *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* $\Pr[|\hat{V} - V| > \varepsilon V] < \delta]$.

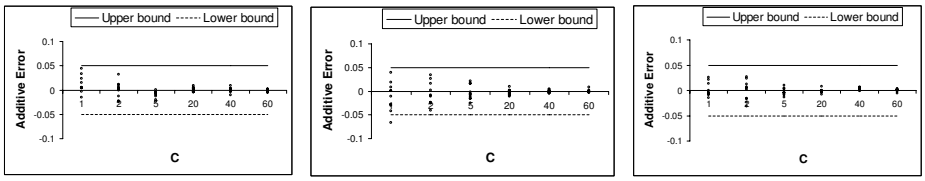
3.5 Experiments

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



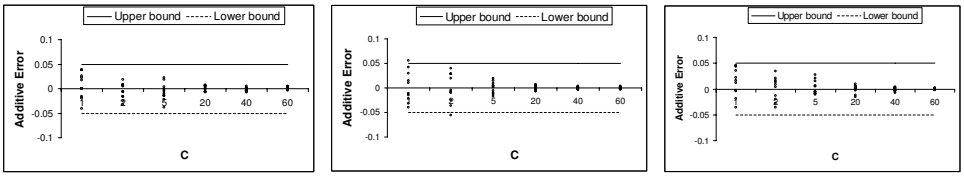
(a) P_1: $P_1(v,w) = 1$ iff $v/w \geq 2$ (b) P_2: $P_2(v,w) = 1$ iff $v/w \geq 3$ (c) P_3: $P_3(v,w) = 1$ iff $v/w \geq 4$

Figure 3.5 Selectivity with Exponential Decay: Accuracy vs C ($\varepsilon = 0.05, \beta = 0.01$)



(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 $v/w \geq 4$

Figure 3.6 Selectivity with Polynomial Decay: Accuracy vs C ($\varepsilon = 0.05, \alpha = 1.0$)



(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 $v/w \geq 4$

Figure 3.7 Selectivity with Sliding Window: Accuracy vs C ($\varepsilon = 0.05$, $W = 200$ seconds)

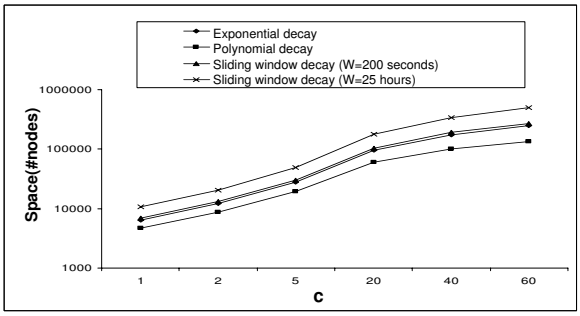
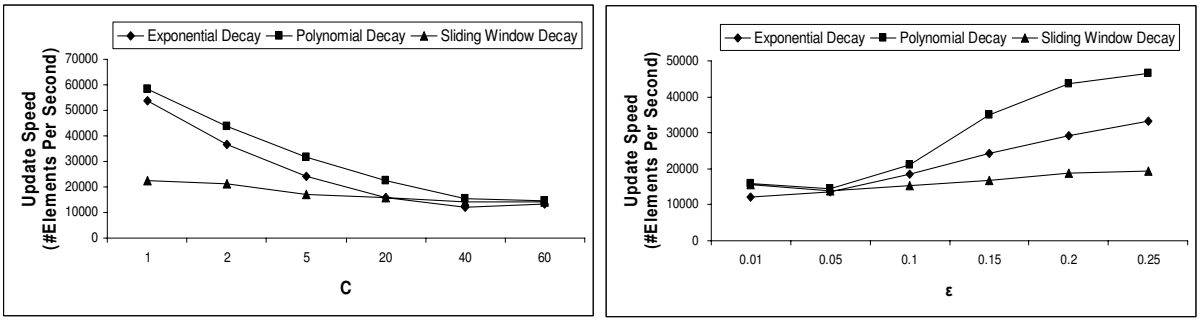


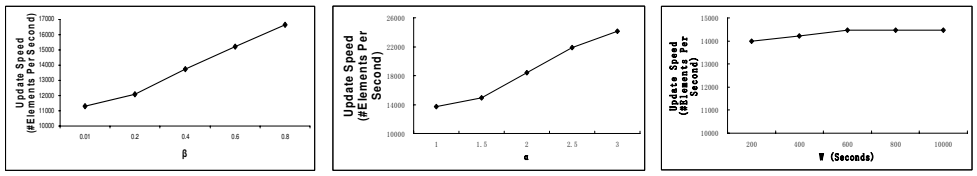
Figure 3.8 Space vs C ($\varepsilon = 0.05$, $\alpha = 1.0$, $\beta = 0.01$, $W = 200$ seconds, 25 hours)



(a) Update Speed vs C
($\varepsilon = 0.05$, $\alpha = 1.0$, $\beta = 0.01$, $W = 200$ seconds)

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

Figure 3.9 Update speed for different decay functions



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

Figure 3.10 Update Speed vs Decay Degree ($\varepsilon = 0.05$, $C = 60$)

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 $\beta \in \{0.01, 0.2, 0.4, 0.6, 0.8\}$

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 3.3.6).

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.

3.5.1 Experimental Setup

We implemented the sketch and the RangeSample algorithm (67) 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 (78), 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 (78) 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.

3.5.2 Accuracy vs Space Usage

Recall that the theoretically derived sample size is $\frac{C}{\varepsilon^2}$ for an ε-approximation (with probability $\geq \frac{2}{3}$) of the time decayed sum ($C = 60$, Theorem 3.3.4) and the time decayed selectivity ($C = 492$, Theo-

rem 3.3.5). 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 3.4, 3.5, 3.6 and 3.7, shows the influence of C on the accuracy of estimations of the sum and the selectivity. In these experiments we set $\varepsilon = 0.05, \alpha = 1$, $\beta = 0.01$ 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 3.4). 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 3.5, 3.6 and 3.7), whereas the y-axis of each dot is the additive error in the estimate for the selectivity.

Figure 3.4, 3.5, 3.6 and 3.7 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 3.8. Besides the exponential decay and polynomial decay, for which α, β are assigned the same values as in Figure 3.4, 3.5 and 3.6, 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 3.4.2). Figure 3.8 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 3.8 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.

3.5.3 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 3.9(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 3.8), 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 3.9(b)).

Both Figures 3.9(a) and 3.9(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 3.10(a) and 3.10(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 3.10(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).

3.6 Concluding Remarks

In this chapter, 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.

CHAPTER 4. General Time-decay Based Correlated Processing

Data stream analysis frequently relies on identifying correlations and posing conditional queries on the data after it has been seen. *Correlated aggregates* form an important example of such queries, which ask for an aggregation over one dimension of stream elements which satisfy a predicate on another dimension. Since recent events are typically more important than older ones, *time decay* should also be applied to downweight less significant values. This chapter presents space-efficient algorithms as well as space lower bounds for the time-decayed correlated sum, a problem at the heart of many related aggregations. By considering different fundamental classes of decay functions, we separate cases where efficient approximations with relative error or additive error guarantees are possible, from other cases where linear space is necessary to approximate. In particular, we show that no efficient algorithms with relative error guarantees are possible for the popular sliding window and exponential decay models, resolving an open problem. This negative result for the exponential decay holds even if the stream is allowed to be processed in multiple passes. The results are surprising, since efficient approximations are known for other data stream problems under these decay models. This is a step towards better understanding which sophisticated queries can be answered on massive streams using limited memory and computation.

4.1 Introduction

There has been much research on estimating aggregates along a single dimension of a stream, such as the median, frequency moments, entropy, etc. However, most streams consist of multi-dimensional data. An example stream of VoIP call data records (CDRs) may have the call start time, end time, and packet loss rate, along with identifiers such as source and destination phone numbers. It is imperative to compute more complex multi-dimensional aggregates, especially those that can “slice and dice” the

data across some dimensions before performing an aggregation, possibly along a different dimension. In this chapter, we consider such *correlated aggregates*, which are a powerful class of queries for manipulating multi-dimensional data. These were motivated in the traditional OLAP model (18), and subsequently for streaming data (6; 40). For example, consider the query on a VoIP CDR stream: "what is the average packet loss rate for calls within the last 24 hours that were less than 1 minute long"? This query involves a selection along the dimensions of call duration and call start time, and aggregation along the third dimension of packet loss rate. Queries of this form are useful in identifying the extent to which low call quality (high packet loss) causes customers to hang up. Another example is: "what is the average packet loss rate for calls started within the last 24 hours with duration greater than the median call length (within the last 24 hours)?", which gives a statistic to monitor overall quality for "long" calls. Such queries cannot be answered by existing streaming systems with guaranteed accuracy, unless they explicitly store all data for the last 24 hours, which is typically infeasible.

In this chapter, we present algorithms and lower bounds for approximating time-decayed correlated aggregates on a data stream. These queries can be captured by three main aspects: selection along one dimension (say x-dimension) and aggregation along a second dimension (say y-dimension) using time-decayed weights defined via a third (time) dimension. The time-decay arises from the fact that in most streams, recent data is naturally more important than older data, and in computing an aggregate, we should give a greater weight to more recent data. In the examples above, the time decay arises in the form of a sliding window of a certain duration (24 hours) over the data stream. More generally, we consider arbitrary time-decay functions which return a weight for each element as a non-increasing function of its age—the time elapsed since the element was generated. Importantly, the nature of the time-decay function will determine the extent to which the aggregate can be approximated.

We focus on the *time-decayed correlated sum* (henceforth referred to as DCS), which is a fundamental aggregate, interesting in its own right, and to which other aggregates can be reduced. An exact computation of the correlated sum requires multiple passes through the stream, even with no time-decay where all elements are weighted equally. Since we can afford only a single pass over the stream, we will aim for approximate answers with accuracy guarantees. In this chapter, we present the first streaming algorithms for estimating the DCS of a stream using limited memory, with such guarantees. Prior work on correlated aggregates either did not have accuracy guarantees on the results (40) or else

did not allow time-decay (6). We first define the stream model and the problem more precisely, and then present our results.

4.1.1 Problem Formulation

Data Stream. We consider an asynchronous stream of (v,w,t) tuples, which is projected on the dimensions of *value*,*weight* and *time* from the stream R defined in Section 1.3. When the context is clear, we still use $R = e_1, e_2, \ldots, e_n$ to represent the projected stream, i.e., $e_i = (v_i, w_i, t_i)$. Let $[m] = \{0, 1, \ldots, m\}$ denote an ordered domain where v_i is drawn from. An example data stream from applications that can be captured by this data stream model is the stream of VoIP call records. There is one stream element per call, where t_i is the time the call was placed, v_i is the duration of the call, and w_i the packet loss rate.

Time decay. We consider aggregates that are time-decayed. The decayed weight of a stream element is returned by a user specified decay function $f(w,x)$, which take as input the initial weight w and age $x = c - t$ of the element, as defined in Section 1.4. Note that c denotes the current time. In this chapter, we only consider decomposable decay functions which has the form of $f(w,x) = w \cdot g(x)$, as defined in Section 1.4.2.

Time-Decayed Correlated Sum. The query for the time-decayed correlated sum over stream R under a prespecified decomposable decay-function g is posed at time t, provides a parameter $\tau \geq 0$, and asks for S^g_τ, defined as follows:

$$S^g_\tau = \sum_{e_i \in R | v_i \geq \tau} w_i \cdot g(t - t_i)$$

A correlated aggregate query could be: "What is the average packet loss rate for all calls which started in the last 24 hours, and were more than 30 minutes in length?". This query can be split into two sub-queries: The first sub-query finds the number of stream elements (v_i, w_i, t_i) which satisfy $v_i > 30$, and $t_i > t - 24$ where t is the current time in hours. The second sub-query finds the sum of w_is for all elements (v_i, w_i, t_i) such that $v_i > 30$ and $t_i > t - 24$. The average is the ratio of the two answers.

DCS lies at the heart of many other aggregates. Example time-decayed aggregates that can reduced to DCS are the following:

- The *time-decayed relative frequency* of a value v, which is given by $(S^g_v - S^g_{v+1})/S^g_0$.

- The *sum of time-decayed weights* of elements in the range $[l,r]$, which is $S^g_l - S^g_{r+1}$.
- The *time-decayed frequency* of range $[l,r]$, which is $(S^g_l - S^g_{r+1})/S^g_0$.
- The *time-decayed ϕ-heavy hitters*, which are all the v's such that the time decayed relative frequency of v is at least ϕ.
- The *time decayed correlated ϕ-quantile*, which is the largest v, such that $(S^g_0 - S^g_v)/S^g_0 \leq \phi$.

Time-Decayed Correlated Count. An important special case of DCS is the *time-decayed correlated count* (henceforth referred to as DCC), where all the weights w_i are assumed to be 1. The correlated count C^g_τ is therefore:

$$C^g_\tau = \sum_{e_i \in R | v_i \geq \tau} g(t - t_i)$$

4.1.2 Contributions

Our main result is that there exist small space algorithms for approximating DCS over an arbitrary decay function g with a small *additive* error. But, the space cost of approximating DCS with a small *relative* error depends strongly on the nature of the decay function—this is possible on some classes of functions using small space, while for other classes, including sliding window and exponential decay, this is provably impossible in sublinear space. More specifically, we show:

1. For *any* decay function g, there is a randomized algorithm for approximating DCS with bounded additive error guarantee which uses space logarithmic in the size of the stream. This significantly improves on previous work (40), which presented heuristics only for sliding window decay. (Section 4.3.1)
2. On the other hand, for any *finite* decay function, defined in Section 1.4.1, we show that approximating DCS with a small *relative* error needs space linear in the size of the elements whose ages are not larger than the age limit of the decay function. Because sliding window decay is a finite decay function, the above two results resolve the open problem posed in (6), which was to determine the space complexity of approximating the correlated sum under sliding window decay. (Section 4.4.1)

3. For any non-exponential converging decay function, defined in Section 1.4.1, there is an algorithm for approximating DCS to within a small relative error using space logarithmic in the stream size, and logarithmic in the "rate" of the decay function. (Section 4.3.2)
4. For any exponential decay function and super-exponential decay function, defined in Section 1.4.1, we show that the space complexity of approximating DCS with a small relative error is linear in the stream size, in the worst case, even if multi-pass processing of the stream is allowed. This may be surprising, since there are simple and efficient solutions for maintaining exponentially decayed sum exactly in the non-correlated case. (Section 4.4.2)

We evaluate our techniques over real and synthetic data in Section 4.5, and observe that they can effectively summarize massive streams in tens of kilobytes.

4.2 Prior Work

Concepts of correlated aggregation in the (non-streaming) OLAP context appear in (18). The first work to propose correlated aggregation for streams was Gehrke *et al.* (40). They assumed that data was locally uniform to give heuristics for computing the non-decayed correlated sum where the threshold (τ) is either an extrema (min, max) or the mean of the all the received values (v_i's). For the sliding window setting, they simply partition the window into fixed-length intervals, and make similar uniformity assumptions for each interval. No strong guarantee on the answer quality are provided by any of these approaches. Subsequently, Ananthakrishna *et al.* (6) presented summaries that estimate the non-decayed correlated sum with *additive error* guarantees. The problem of tracking sliding window based correlated sums with quality guarantees was given as an open problem in (6). We show that this relative error guarantees are not possible while using small space, whereas additive guarantees can be obtained.

Xu *et al.* (79) proposed the concept of asynchronous streams. They gave a randomized algorithm to approximate the sum and the median over sliding windows. Busch and Tirthapura (14) later gave a deterministic algorithm for the sum. Cormode *et al.* (30; 25) gave algorithms for general time decay based aggregates over asynchronous streams. By defining timestamps appropriately, *non-decayed* correlated sum can be reduced to the sum of elements within a sliding window over an asynchronous

stream. As a result, relative error bounds follow from bounds in (79; 14; 30; 25). But these methods do not extend to accurately estimating DCS or DCC.

Datar *et al.* (35) presented a bucket-based technique called *exponential histograms* for sliding windows on synchronous streams. This approximates counts and related aggregates, such as sum and ℓ_p norms. Gibbons and Tirthapura (42) improved the worst-case performance for counts using a data structure called a *wave*. Going beyond sliding windows, Cohen and Strauss (21) formalized time-decayed data aggregation, and provided strong motivating examples for non-sliding window decay. All these works emphasized the time decay issue, but did not consider the problems of correlated aggregate computation.

4.3 Upper Bounds

In this section, we present algorithms for approximating DCS over a stream R. The main results are: (1) For an arbitrary decay function g, there is a small space streaming algorithm to approximate S_τ^g with a small additive error. (2) For any non-exponential *converging* decay function g, there is small space streaming algorithm to approximate S_τ^g with a small relative error.

4.3.1 Additive Error

A predicate $P(v,w)$ is a binary function of v and w, used to select certain items. For example, (1) the predicate could select only those items with $v > 1000$ by returning 1 for those items, and 0 for others; (2) the predicate could select only those items with $w < 100$ similarly; and (3) the predicate could select only those items with both $v > 1000$ and $w < 100$, and so on. The time-decayed selectivity Q of a predicate $P(v,w)$ on a stream R of (v,w,t) tuples is defined as

$$Q = \frac{\sum_{(v,w,t)\in R} P(v,w)\cdot w\cdot g(c-t)}{\sum_{(v,w,t)\in R} w\cdot g(c-t)}$$

The decayed sum S is defined as:

$$S = \sum_{(v,w,t)\in R} w\cdot g(c-t)$$

Note that $S = S_0^g$. We use the following results on time-decayed selectivity estimation from Chapter 3 in our algorithm for approximating DCS with a small additive error.

Theorem 4.3.1 (Theorems 3.4.1, 3.4.2, 3.4.3)**.** *Given $0 < \varepsilon < 1$ and probability $0 < \delta < 1$, there exists a small space sketch of size $O((1/\varepsilon^2) \cdot \log(1/\delta) \cdot \log M)$ that can be computed in one pass from stream R, where M is an upper bound on S. For any decay function g given at query time: (1) the sketch can return an estimate $\widehat{S}$ for S such that $\Pr[|\widehat{S} - S| \leq \varepsilon S] \geq 1 - \delta$. (2) Given predicate $P(v,w)$ at query time, the sketch gives an estimate $\widehat{Q}$ for the decayed selectivity Q, such that $\Pr[|\widehat{Q} - Q| \leq \varepsilon] \geq 1 - \delta$.*

The sketch designed in Chapter 3 can be thought of as computing a set of fixed-sized random samples of the stream. Each successive sample is chosen with decreasing probability, so for a unit-weight stream element, its probability of selection in each successive sample is $1, \frac{1}{2}, \frac{1}{4}, \ldots$. For non-unit weight elements, the probability of selecting a stream element into the sample is also proportional to the decayed weight of the element. The result stated in the above theorem allows DCS to be additively approximated:

Theorem 4.3.2. *For an arbitrary decay function g, there exists a small space sketch of R that can be computed in one pass over the stream. At any time instant, given a threshold τ, the sketch can return $\widehat{S}_\tau^g$, such that $|\widehat{S}_\tau^g - S_\tau^g| \leq \varepsilon S_0^g$ with probability at least $1 - \delta$. The space complexity of the sketch is $O((1/\varepsilon^2)\log(1/\delta) \cdot \log M)$, where M is an upper bound on S_0^g.*

Proof. We run the sketch algorithm in Chapter 3 on stream R, with approximation error $\varepsilon/3$ and failure probability $\delta/2$. Let this sketch be denoted by $\mathscr{K}$. Where the function g is implicit, we can drop it from our notation, and simply write $\widehat{S}_\tau$, S_τ in place of $\widehat{S}_\tau^g$, S_τ^g respectively.

Given τ at query time, we define a predicate P for the selectivity estimation as: $P(v,w) = 1$, if $v \geq \tau$, and $P(v,w) = 0$ otherwise. The selectivity of P is $Q = S_\tau/S$. Then $\mathscr{K}$ can return estimates $\widehat{Q}$ of Q and $\widehat{S}$ of S such that

$$\Pr\left[|\widehat{Q} - Q| > \frac{\varepsilon}{3}\right] \leq 1 - \frac{\delta}{2} \tag{4.1}$$

$$\Pr\left[|\widehat{S} - S| > \frac{\varepsilon S}{3}\right] \leq 1 - \frac{\delta}{2} \tag{4.2}$$

Our estimate $\widehat{S}_\tau$ is given by $\widehat{S}_\tau = \widehat{S} \cdot \widehat{Q}$. From (4.1) and (4.2), and using the union bound on probabilities,

we get that the following events are both true, with probability at least $1-\delta$.

$$Q-\frac{\varepsilon}{3} \leq \widehat{Q} \leq Q+\frac{\varepsilon}{3} \tag{4.3}$$

$$S\left(1-\frac{\varepsilon}{3}\right) \leq \widehat{S} \leq S\left(1+\frac{\varepsilon}{3}\right) \tag{4.4}$$

Using the above, and using $Q = S_\tau/S$, we get

$$\widehat{S_\tau} \leq \left(\frac{S_\tau}{S}+\varepsilon/3\right)\cdot S\cdot(1+\varepsilon/3) = S_\tau + \frac{S_\tau\varepsilon}{3} + S\left(\frac{\varepsilon}{3}+\frac{\varepsilon^2}{9}\right) \leq S_\tau + \varepsilon S$$

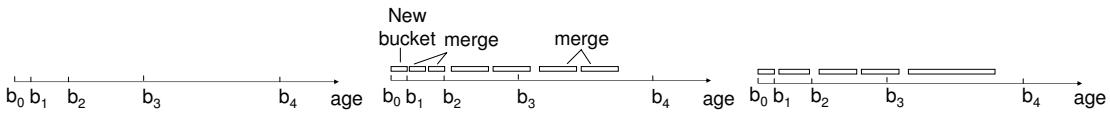
In the last step of the above inequality, we have used the fact $S_\tau \leq S$ and $\varepsilon < 1$. Similarly, we get that if (4.3) and (4.4) are true, then, $\widehat{S_\tau} \geq S_\tau - \varepsilon S$, thus completing the proof that $\mathscr{K}$ can (with high probability) provide an estimate $\widehat{S^g_\tau}$ such that $|\widehat{S^g_\tau} - S^g_\tau| \leq \varepsilon S^g_0$ □

An important feature of this approach, made possible due to the flexibility of the sketch in Theorem 4.3.1, is that it allows the decay function g to be specified at query time, i.e. after the stream R has been seen. This allows for a variety of decay models to be applied in the analysis of the stream after the fact. Further, since the sketch is designed to handle asynchronous arrivals, the timestamps can be arbitrary and arrivals do not need to be in timestamp order.

4.3.2 Relative Error

In this section, we present a small space sketch that can be maintained over a stream R with the following properties. For an arbitrary *converging* decay function g (defined in Section 1.4.1), which is known beforehand, and a parameter τ which is provided at query time, the sketch can return an estimate $\widehat{S^g_\tau}$ which is within a small relative error of S^g_τ. The space complexity of the sketch depends on g.

The idea behind the sketch is to maintain multiple data structures each of which solves the undecayed correlated sum, and partition stream elements across different data structures, depending on their timestamps, following the approach of the Weight-Based Merging Histogram (WBMH), due to Cohen and Strauss (21). In the rest of this section, we first give high level intuition, followed by a formal description of the sketch, and a correctness proof. Finally, we describe enhancements that allow faster insertion of stream elements into the sketch.



(a) Regions (b) Buckets & Regions (c) Buckets & Regions after Merge

Figure 4.1 Weight-based merging histograms.

4.3.2.1 Intuition

We first describe the weight-based merging histogram (21). The histogram partitions the stream elements into buckets based on their age. Given a decay function g, and parameter ε_1, the sequence $b_i, i \geq 0$ is defined as follows: $b_0 = 0$, and for $i > 0$, b_i is defined as the largest integer such that $g(b_i - 1) \geq g(b_{i-1})/(1+\varepsilon_1)$ (Figure 4.1(a)).

For simplicity, we first describe the algorithm for the case of a strictly synchronous stream, where the timestamp of a stream element is just its position in the stream. We later discuss the extension to asynchronous streams. Let G_i denote the interval $[b_i, b_{i+1})$ so that $|G_i| = b_{i+1} - b_i$. Once the decay function is given, the G_is are fixed and do not change over time. The elements of the stream are grouped into regions based on their age. For $i \geq 0$, region i contains all stream elements whose age lies in interval G_i.

For any i, we have $g(b_i) < g(b_0)/(1+\varepsilon_1)^i$, and thus we get $i < \log_{1+\varepsilon_1}(g(0)/g(b_i))$. Since the age of an element cannot be more than n, $b_i \leq n$. Thus we get that the total number of regions is no more than $\beta = \lceil \log_{1+\varepsilon_1}(g(0)/g(n)) \rceil$. From the definition of the b_is, we also have the following fact.

Fact 4.3.1. *Suppose two stream elements have ages a_1 and a_2 so that a_1 and a_2 fall within the same region. Then,*

$$\frac{1}{1+\varepsilon_1} \leq \frac{g(a_1)}{g(a_2)} \leq 1+\varepsilon_1$$

The data structure maintains a set of *buckets*. Each bucket groups together stream elements whose timestamps fall in a particular range, and maintains a small space summary of these elements. We say that the bucket is "responsible" for this range of timestamps (or equivalently, a range of ages).

Suppose that the goal was to maintain S_0^g, just the time-decayed sum of all stream elements. If the current time c is such that $c \mod b_1 = 0$, then a new bucket is created for handling future elements (Figure 4.1(b)). The algorithm ensures that the number of buckets does not grow too large through

the following rule: if two adjacent buckets are such that the age ranges that they are responsible for are both contained within the same region, then the two buckets are merged into a single bucket. The count within the resulting bucket is equal to the sum of the counts of the two buckets, and the resulting bucket is responsible for the union of the ranges of timestamps the two buckets were responsible for (Figure 4.1(c)).

Due to the merging, there can be at most 2β buckets: one bucket completely contained within each region, and one bucket straddling each boundary between two regions. From Fact 4.3.1, the weights of all elements contained within a single bucket are close to each other, and since g is a converging decay function, this remains true as the ages of the elements increase. Consequently, WBMH can approximate S_0^g with ε_1 relative error by treating all elements in each bucket as if they shared the smallest timestamp in the range, and scaling the corresponding weight by the total count.

However, this does not solve the more general DCS problem, since it does not allow filtering out elements whose values are smaller than τ. We extend the above data structure to the DCS problem by embedding within each bucket a data structure that can answer the (undecayed) correlated sum of all elements that were inserted into this bucket. This data structure can be any of the algorithms that can estimate the sum of elements within a sliding window on asynchronous streams, including the sketch designed in Chapter 2 and in (25; 14): values of the elements are treated as timestamps, and a window size $m - \tau + 1$ is supplied at query time (where m is an upper bound on the value).

These observations yield our new algorithm for approximating S_τ^g. We replace the simple count for each bucket in the WBMH with a small space sketch, from either one designed in Chapter 2 or in (25). We will not assume a particular sketch for maintaining the information within a bucket. Instead, our algorithm will work with any sketch that satisfies the following properties—we call such a sketch a "bucket sketch". Let ε_2 denote the accuracy parameter for such a bucket sketch.

1. The bucket sketch must concisely summarize a stream of (v, w) pairs using space polylogarithmic in the stream size. Given parameter $\tau \geq 0$ at query time, the sketch must return an estimate for $\sum_{v \geq \tau} w$, such that relative error of the estimate is within ε_2.
2. It must be possible to merge two bucket sketches easily into a single sketch. More precisely, suppose that S_1 is the sketch for a set of elements R_1 and S_2 is the sketch for a set of elements R_2, then it must be possible to merge together S_1 and S_2 to get a single sketch denoted by $S = S_1 \cup S_2$,

such that S retains Property 1 for the set of elements $R_1 \cup R_2$.

The analysis of the sketch proposed in Chapter 2 explicitly shows that the above properties hold. Likewise, the sketch designed in (25) also has the necessary properties, since it is built on with multiple instances of q-digest summaries (71) which are themselves mergable. The different sketches have slightly different time and space complexities; we state and analyze our algorithm in terms of a generic bucket sketch, and subsequently describe the cost depending on the choice of sketch.

4.3.2.2 Formal Description and Correctness

Recall that ε is the required bound on the relative error. Our algorithm combines two data structures: the WBMH with accuracy parameter $\varepsilon_1 = \varepsilon/2$; and the bucket sketches with accuracy parameter $\varepsilon_2 = \varepsilon/2$. The initialization is shown in the SETBOUNDARIES procedure (Algorithm 14), which creates the regions G_i by selecting $b_0, \ldots, b_\beta$. For simplicity of presentation, we have assumed that the maximum stream length n is known beforehand, but this is not necessary — the b_i's can be generated incrementally, i.e., b_i does not need to be generated until element ages exceeding b_{i-1} have been observed.

Algorithm 15 shows the PROCESSELEMENT procedure for handling a new stream element. Whenever the current time t satisfies $t \mod b_1 = 0$, we create a new bucket to summarize the elements with timestamps from t to $t + b_1 - 1$ and seal the last bucket which was created at time $t - b_1$. The procedure FINDREGIONS(t) returns the set of regions that contain buckets to be merged at time t. In the next section we present novel methods to implement this requirement efficiently. Algorithm 16 shows the procedure RETURNAPPROXIMATION which generates the answer for a query for S^g_τ at time t. Each bucket returns an estimate for the total undecayed weights of the elements that were inserted in the bucket and whose values are not smaller than τ. Each of these estimates is then scaled down by a factor of the decay function value using the corresponding $t - F_B$ as the parameter of the decay function. The summation of these scaled estimates are returned as the estimate for S^g_τ.

Theorem 4.3.3. *If g is a converging decay function, for any τ given at any time t, the algorithm specified in Algorithm 14, 15 and 16 can return $\widehat{S}^g_\tau$, such that $(1-\varepsilon)S^g_\tau \leq \widehat{S}^g_\tau \leq (1+\varepsilon)S^g_\tau$.*

Proof. For the special converging decay function where $g(x) \equiv 1$ (no decay), then WBMH has only

Algorithm 14: SetBoundaries(ε)

Task: create $G_0, G_1, \ldots, G_\beta$ using $\varepsilon_1 = \varepsilon/2$ to initialize regions.

1 $b_0 \leftarrow 0$;
2 **for** $1 \le i \le \beta$ **do** $b_i \leftarrow \max_x\{x|(1+\frac{\varepsilon}{2})g(x-1) \ge g(b_{i-1})\}$; `/* x are integers */`
3 $j \leftarrow -1$; `/* j is the index of the active bucket for new elements */`

one region and one bucket. So the algorithm reduces to a single bucket sketch. This sketch can directly provide an $\varepsilon_2 = \varepsilon/2$ relative error guarantee for the estimate of S^g_τ.

The broader case is where $g(x+1)/g(x)$ is non-decreasing with x. Let $\{B_1, \ldots, B_k\}$ be the set of sketch buckets at query time t. Let $R_i \subseteq R$ be the substream that was inserted into B_i, $1 \le i \le k$. Since every stream element is inserted into exactly one sketch bucket at any time, the R_is partition R: $\bigcup_{i=1}^k R_i = R$ and $R_i \cap R_j = \emptyset$ if $i \ne j$. Note that merging two buckets just creates a new bucket for the union of the two underlying substreams. Let $S^g_{\tau,i} = \sum_{e_j \in R_i | v_j \ge \tau} w_j g(t-j)$ be the DCS of R_i at time t, $1 \le i \le k$, so $S^g_\tau = \sum_{i=1}^k S^g_{\tau,i}$. We first consider the accuracy of the estimate for each $S^g_{\tau,i}$ using sketch bucket B_i, $1 \le i \le k$.

For each $(v_j, w_j, j) \in R_i$ at any query time t, since $F_{B_i} \le j \le L_{B_i}$ which implies $g(t-F_{B_i}) \le g(t-j) \le g(t-L_{B_i})$, and $g(t-L_{B_i})/(1+\varepsilon_1) \le g(t-F_{B_i})$ (using Fact 4.3.1), we have

$$\frac{w_j g(t-j)}{1+\varepsilon_1} \le \frac{w_j g(t-L_{B_i})}{1+\varepsilon_1} \le w_j g(t-F_{B_i}) \le w_j g(t-j)$$

therefore,

$$\frac{1}{1+\varepsilon_1} \sum_{e_j \in R_i | v_j \ge \tau} w_j g(t-j) \le g(t-F_{B_i}) \sum_{e_j \in R_i | v_j \ge \tau} w_j \le \sum_{e_j \in R_i | v_j \ge \tau} w_j g(t-j)$$

i.e.,

$$\frac{1}{1+\varepsilon_1} S^g_{\tau,i} \le g(t-F_{B_i}) Q_i \le S^g_{\tau,i}\text{ , where } Q_i = \sum_{e_j \in R_i | v_j \ge \tau} w_j \tag{4.5}$$

Also since sketch bucket B_i can return $\widehat{Q}_i$ such that (Chapter 2 or (25))

$$(1-\varepsilon_2) Q_i \le \widehat{Q}_i \le (1+\varepsilon_2) Q_i \tag{4.6}$$

Algorithm 15: ProcessElement(v_i, w_i, i)

Task: Insert a new element

```
1  if i mod b_1 = 0 then
2      j ← j + 1;
3      Initialize a new bucket sketch B_j with accuracy ε/2;
4      F_{B_j} ← i;
5      L_{B_j} ← i + b_1 − 1 ;                    /* Set timestamp range covered by B_j */
6  Insert (v_i, w_i) into B_j;
7  foreach g ∈ FINDREGIONS(i) do  /* Set of regions with buckets to be merged at
   time i */
8      b_min ← min_t{t|t ∈ G_g};
9      b_max ← max_t{t|t ∈ G_g} ;     /* Find the left and right timestamp boundary of
       region G_g */
10     Find buckets B' and B'', such that b_min ≤ (i − L_{B'}) < (i − F_{B'}) < (i − L_{B''}) < (i − F_{B''}) ≤ b_max ;
       /* Find buckets covered by G_g */
11     B ← B' ∪ B'' ;                                  /* merge two buckets */
12     F_B ← F_{B''};
13     L_B ← L_{B'};
14     Drop B' and B''
```

Combining 4.5 and 4.6, we have

$$\frac{1-\varepsilon_2}{1+\varepsilon_1} S^g_{\tau,i} \leq \widehat{Q}_i \cdot g(t - F_{B_i}) \leq (1+\varepsilon_2) S^g_{\tau,i}.$$

Now we sum all the $S^t_{\tau,i}$ together, $i = 1, 2, \ldots, k$, we get

$$\frac{1-\varepsilon_2}{1+\varepsilon_1} \sum_{i=1}^{k} S^g_{\tau,i} \leq \sum_{i=1}^{k} \widehat{Q}_i \cdot g(t - F_{B_i}) \leq (1+\varepsilon_2) \sum_{i=1}^{k} S^g_{\tau,i}.$$

i.e.,

$$\frac{1-\varepsilon_2}{1+\varepsilon_1} S^g_{\tau} \leq \widehat{S^g_{\tau}} \leq (1+\varepsilon_2) S^g_{\tau}$$

Using the facts $0 < \varepsilon < 1$ and $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$, we get the stated accuracy guarantee. □

Algorithm 16: ReturnApproximation(τ, t)

Task: Return an estimate for S_τ^g

```
1 Let the set of buckets be: {B_1, B_2, ..., B_k} ;        /* for some k, 1 ≤ k ≤ 2β */
2 s ← 0;
3 for 1 ≤ i ≤ k do
4     Let Q̂_i be result for B_i using m − τ + 1 as window size;
5     s ← s + Q̂_i · g(t − F_{B_i}) ;  /* Approx sum of element weights in B_i with v_i ≥ τ */
6 return Ŝ_τ^g = s;
```

4.3.2.3 Fast Bucket Merging

At every clock tick the WBMH maintenance algorithm needs to check whether there are buckets that need to be merged. A naive solution is to go through all the buckets and merge those that are covered by the same region. This procedure can severely reduce the speed of stream processing. In this section we present an algorithm which, given the clock time t, can efficiently return the set of regions that have buckets to be merged at time t.

Definition 4.3.1 (Sketch bucket B's capacity $|B|$)**.** *The capacity of bucket B is given by $|B| = L_B - F_B + 1$, where L_B and F_B are the largest and smallest timestamps of the elements that were inserted into B (as in Algorithm 15).*

Definition 4.3.2 (Bucket capacity in the ith region)**.** *Define $I_0 = 1$. For $0 < i < \beta$, let $I_i = |B|$, where B is any bucket such that $t - F_B = b_i$ for some value of t.*

In the next lemma, we show each I_i is a constant, $0 < i < \beta$, and can be directly computed.

Lemma 4.3.1. *For $0 < i < \beta$, $I_i = \lfloor |G_{i-1}|/I_{i-1} \rfloor \cdot I_{i-1}$*

Proof. The lemma is proved by induction. For the base case, since the capacity of the new bucket created in G_0 is exactly equal to $|G_0|$, no merges can happen in G_0, so $I_1 = |G_0| = \lfloor |G_0|/I_0 \rfloor \cdot I_0$. For the inductive step, suppose the claim is true for some i, i.e,, I_i is a constant, which implies at most $\lfloor |G_i|/I_i \rfloor$ buckets of capacity I_i can be merged within G_i. The new bucket from merging therefore has capacity $\lfloor |G_i|/I_i \rfloor I_i = I_{i+1}$, which is a constant. This completes the proof. □

In the next lemma, we show that given I_i we can directly find the sequence of time points at which G_i has buckets to be merged.

Algorithm 17: InitializeFindRegions()

Task: Initialize hash table with merging times.

```
1 Initialize hash table T;
2 I_0 ← 1;
3 for 1 ≤ i ≤ β − 1 do I_i ← ⌊|G_{i−1}|/I_{i−1}⌋ I_{i−1} ;        /* From Lemma 4.3.1 */
4 for 1 ≤ i ≤ β − 1 do
5     if ⌊|G_i|/I_i⌋ ≥ 2 then Insert (i, b_i + 2I_i) into hash table T ;   /* Compute the time at
      which G_i firstly has mergable buckets */
```

Lemma 4.3.2. *For each $i \in \{j | j = 0$ or $\lfloor |G_j|/I_i \rfloor < 2\}$, G_i has no buckets to be merged at any time; for each $i \in \{j | j > 0$ and $\lfloor |G_j|/I_i \rfloor \geq 2\}$, G_i has buckets to be merged at time $\{b_i + (k\lfloor |G_i|/I_i \rfloor + j)I_i\}$, for each j and k, $2 \leq j \leq \lfloor |G_i|/I_i \rfloor$, $k \geq 0$.*

Proof. The new bucket created in G_0 has capacity equal to $|G_0|$, so G_0 does not have any buckets to be merged at any time. For $i > 0$, if $\lfloor |G_i|/I_i \rfloor < 2$, then G_i will not have the chance to have two buckets of capacity I_i to be merged at any time. Now we consider the case where $\lfloor |G_i|/I_i \rfloor \geq 2$ and $i > 0$. G_i has its first whole bucket at time $t = b_i + I_i$. Note that within G_i at most $\lfloor |G_i|/I_i \rfloor$ buckets that enter G_i can be merged together. Thus, (1) at time $t = b_i + 2I_i, b_i + 3I_i, \ldots, b_i + \lfloor |G_i|/I_i \rfloor \cdot I_i$, buckets can be merged within G_i; (2) This sequence of merge operations repeats every $\lfloor |G_i|/I_i \rfloor \cdot I_i$ clock ticks, meaning G_i has buckets to be merged at times $\{b_i + (k\lfloor |G_i|/I_i \rfloor + j)I_i\}$, for each j and k, $2 \leq j \leq \lfloor |G_i|/I_i \rfloor$, $k \geq 0$. □

Lemma 4.3.2 provides a way for each region to directly compute the sequence of time points at which it has buckets to be merged. Based on this observation, we present the algorithm that given time t returns the set of regions, which have buckets to be merged at time t.

Algorithm for Fast Bucket Merging. Our implementation of the algorithm uses a hash table T to store the set of buckets that need to be merged at timestamp t. In particular, t is hashed to the index of a table cell which stores the set of (i,t) pairs, such that region G_i has buckets to be merged at time t. Algorithm 17 shows procedure INITIALIZEFINDREGIONS() which first computes I_i using Lemma 4.3.1. It then uses Lemma 4.3.2 to fill in the earliest time at which region G_i will have buckets to be merged. At time t, FINDREGIONS(t) (Algorithm 18) retrieves the set of regions that have buckets to be merged, and deletes those regions from the hash table. Then, for each returned region, we compute its next merging time using Lemma 4.3.2 and store the results into the corresponding hash table cells

Algorithm 18: FindRegions(t)

Task: Find mergable regions at time t.

```
M ← ∅;
foreach (i,t) ∈ T do          /* Region G has buckets to be merged at time t */
    M ← M ∪ {i};
    if (t − b_i)/I_i mod ⌊|G_i|/I_i⌋ = 0 then t' ← t + 2I_i;
    else t' ← t + I_i ;  /* Find the next time at which G_i has mergable buckets */
    Insert (i,t') into hash table T;
return M ;          /* set of regions with buckets to be merged at time t */
```

for the future lookup.

4.3.2.4 Time and Space Complexity

The time complexity depends on the sketch bucket that we chose and the decay function g given by the user.

Theorem 4.3.4. *The (amortized) time complexity of the algorithm per update in Algorithm 15 is* $O(Q(M/n)+\log Q)$*, where* M *is the total number of merges happened in processing the stream, and*

1. $Q = O\left(\frac{1}{\varepsilon^2}\log\frac{\beta}{\delta}\log n\right)$ *is the size of the sketch bucket in words in Chapter 2.*
2. $Q = O\left(\frac{1}{\varepsilon}\log\left(\frac{\varepsilon n}{\log n}\right)\right)$ *is the size of the sketch bucket in words in (25).*

Proof. The per update cost is dominated by: (1) inserting the new element into the bucket, which takes time sublinear in the size of the sketch bucket: $\log Q$. (2) merging buckets when necessary, which can be carried out in time linear in the size of the bucket data structure (Chapter 2 or (25)), so the amortized time for merge per update is $O(Q(M/n))$ (3) Updating the hash table, which has to be done once for every merge that occurs, and takes constant time. Combining these costs leads to the stated time complexity. □

Time dependence on decay function g. As stated in Theorem 4.3.4, the time complexity depends on the value of M, which in turn is determined by the choice of decay function g, since it defines the size of each region in the WBMH and hence the sequence of bucket merges during the stream processing. We show the consequence for various broad classes of decay function:

- In the case of no decay ($g(x) \equiv 1$), the region G_0 is infinitely large, so the algorithm maintains only one bucket and therefore no bucket merges will happen, i.e., $M = 0$, giving the time cost $O(\log Q)$.
- For exponential decay functions $g(x) = 2^{-\alpha x}$, $\alpha > 0$, since all the regions have the same size $|G_i| = \left\lfloor \frac{1}{\alpha} \log_2 \left(1 + \frac{\varepsilon}{2}\right) \right\rfloor$, $0 \le i \le \beta$, no bucket merges will happen, i.e., $M = 0$, giving the time cost $O(\log Q)$.
- For all other decay functions, such as polynomial decay $g(x) = (x+1)^{-a}$, $a > 0$, many bucket merges can happen. For a synchronous stream, there can be at most n bucket merges, as each merge conceptually places two adjacent stream elements which were in different buckets in the same bucket. Thus, whatever the decay function, the total number of merges cannot be larger than the stream size n, i.e., $M \le n$. So the amortized time cost $O(Q)$.

The space complexity includes the space cost for the buckets in the histogram and the hash table. The space to represent each bucket depends on the choice of the bucket sketch.

Theorem 4.3.5. *The space complexity of Algorithm 14, 15 and 16 is $O(\beta(Z + \log n))$ bits, where*

1. $\beta = \left\lceil \log_{1+\varepsilon/2}(g(0)/g(n)) \right\rceil$
2. $Z = O\left(\frac{1}{\varepsilon^2} \log \frac{\beta}{\delta} \log n \log m\right)$ *is the size in bit of the bucket sketch designed in Chapter 2.*
3. $Z = O\left(\frac{1}{\varepsilon} \log m \log\left(\frac{\varepsilon n}{\log n}\right)\right)$ *is the size of the bucket sketch in bits in (25).*

Proof. The number of buckets used is at most 2β. For the randomized sketch designed in Chapter 2, in order to have a δ failure probability bound, by the union bound, we need to set the failure probability for each bucket to be $\delta/(2\beta)$, so we get $Z = O\left(\frac{1}{\varepsilon^2} \log \frac{\beta}{\delta} \log n \log m\right)$ (Lemma 2.2.11). For the deterministic sketch designed in (25), $Z = O\left(\frac{1}{\varepsilon} \log m \log\left(\frac{\varepsilon n}{\log n}\right)\right)$ (Section 3.1 in (25)). The size of the hash table can be set to $O(\beta)$ cells, because each of the β regions occupies at most one cell. Each cell uses $O(\log n)$ bits of space to store the region's index and the region's next merge time. So all together, the total space cost is $O(\beta(Z + \log n))$. □

Space dependence on decay function g. As shown in Theorem 4.3.5, the space complexity depends on the decay function g, since it determines the number of regions (implicitly the number of buckets) in WBMH. We show the consequence for various broad classes of decay function:

- For exponential decay functions $g(x) = 2^{-\alpha x}$, $\alpha > 0$, we have $\beta = \alpha n \log_{1+\varepsilon/2} 2$ and therefore the space complexity is $O(n(\log m)\log n)$ bits. This means that this algorithm needs space linear in the input size.

- For polynomial decay functions $g(x) = (x+1)^{-a}$, $a > 0$, since $\beta = a \log_{1+\varepsilon/2} n$, the space complexity is sublinear, $O\left(\frac{a}{\varepsilon^3}\log^2 n \log m \log\frac{\beta}{\delta}\right)$ using the sketch in Chapter 2, and $O(\frac{a}{\varepsilon^2}\log n \log m \log(\varepsilon n/\log n) + \log^2 n)$ using the sketch of (25);

- In the case of no decay ($g(x) \equiv 1$), the region G_0 is infinitely large, so the algorithm maintains only one bucket, giving space cost $O(Z + \log n)$.

Intuitively the algorithm can approximate S_τ^g with a relative error bound using small space if g decays more slowly than the exponential decay. Further, the space decreases the "slower" that g decays, the limiting case being that of no decay. We complement this observation with the result that the DCS problem under exponential decay requires linear space in order to provide relative error guarantees.

4.3.2.5 Asynchronous Streams

So far our discussion of the algorithm for relative error has focused on the case of strictly synchronous streams, where the elements arrive in order of timestamps. In an asynchronous setting, a new element (v_1, w_1, t_1) may have timestamp $t_1 < t$ where t is the current time. But this can easily be handled by the algorithm described above: the new element is just directly inserted into the earlier bucket which is responsible for timestamp t_1. Meanwhile, at every clock tick t, if no new element with timestamp t is received, we can still maintain the WBMH in the way as if we received a new *dummy* stream element whose timestamp is t, but we do not insert the dummy element into the WBMH. In other words, we create a new sketch bucket for the dummy element when necessary but do not insert it into the sketch bucket (leaving the new sketch bucket empty), and merge all the sketch buckets determined. Therefore, WBMH can be maintained exactly in the same way as in the case where a strictly synchronous stream is processed in Algorithm 15. The accuracy and space guarantees do not alter, although the time cost is affected because for each new element, we need to find the right bucket to insert it.

Let Q, Z, M be the same as defined in Theorem 4.3.4 and 4.3.5. Let L denote the age of the oldest stream element.

Corollary 4.3.5.1. *The (amortized) time complexity of the algorithm per timestep for an asynchronous stream is* $O(Q(M/L)+(n/L)(\log Q+\log\beta))$. *The space complexity of the algorithm for an asynchronous stream is* $O(\beta(Z+\log L))$ *bits, where* $\beta=\left\lceil\log_{1+\varepsilon/2}(g(0)/g(L))\right\rceil$.

Proof. Time complexity. Note that the number of sketch buckets only depends on the decay function and the timestamp range in the stream, and there are no more than $2\beta=2\left\lceil\log_{1+\varepsilon/2}(g(0)/g(L))\right\rceil$ sketch buckets. All the sketch buckets can be managed by a balanced binary search tree with the timestamp ranges of the buckets being the keys, so the time cost in finding the bucket for the insertion of a new element is $O(\log\beta)$. Inserting a new element into a sketch bucket costs time $O(\log Q)$. So the amortized time for inserting elements into WBMH per timestep is $O((n/L)(\log Q+\log\beta)))$. Adding the amortized time cost $O(Q(M/L))$ in merging buckets per timestep, we get the stated time complexity.

Space complexity. The space cost includes the space usage by the sketch buckets $O(\beta Z)$ and the space usage by the hashtable $O(\beta\log L)$. Add them together, we get the stated space complexity. □

We note that in the case where the stream size is relatively much smaller than the timestamp range in the stream, the actual space cost by our algorithm will be much smaller than the (worst case) space complexity stated in the above theorem, because in that case most of the sketch buckets will either be empty or only have a few elements inserted.

4.4 Lower Bounds

This section shows large space lower bounds for finite decay or (super) exponential decay for DCC on strictly synchronous streams. Since DCC is a special case of DCS, and every synchronous stream is also an asynchronous stream, these lower bounds also apply to DCS on asynchronous streams.

4.4.1 Finite Decay

Finite decay, defined in Section 1.4.1, captures the case when after some age limit N, the decayed weight is zero.

Theorem 4.4.1. *For any finite decay function g with age limit N, any streaming algorithm (deterministic or randomized) that can provide an estimate* $\widehat{C}_\tau^g$ *such that* $|\widehat{C}_\tau^g-C_\tau^g|<\varepsilon C_\tau^g$ *for any* τ *given at query*

time for a stream of elements drawn from a universe of size m must require $\Omega(N\log(m/N))$ *bits of space.*

Proof. The bound follows from the hardness of finding the maximum element within a sliding window on a stream of integers. Tracking the maximum within a sliding window of size N over a data stream needs $\Omega(N\log(m/N))$ bits of space, where m is the size of the universe from which the stream elements are drawn (Section 7.4 of (35)).

We show that if there exists an algorithm to approximate $\widehat{C}_\tau^g$, where g has age limit N, then there is an algorithm to find the maximum of the last N elements in R, using the same space. Let α denote the value of the maximum element in the last N elements of the stream. By definition, the decayed weights of the N most recent elements are positive, while all older elements have weight zero.

Note that C_τ^g is a non-increasing function of τ, so $C_\tau^g \geq C_\alpha^g$ for any $\tau < \alpha$. Further, so $C_\alpha^g > 0$, and $C_\tau^g = 0$ for $\tau > \alpha$. If C_τ^g can be approximated with a good relative error, then it is possible to distinguish between the cases $C_\tau^g > 0$ and $C_\tau^g = 0$, for each value of τ. By repeatedly querying the data structure for C_τ^g for different values of τ, we find a value τ^* such that $C_{\tau^*}^g > 0$ and $C_{\tau^*+1}^g = 0$. Then τ^* must be α, the maximum element of the last N elements. □

Since sliding window decay is a special case of finite decay, this shows that approximating C_τ^g with g being a sliding window decay function cannot be solved with relative error in sublinear space. This resolves an open problem identified in (6).

4.4.2 Exponential Decay

Exponential decay functions $g(x) = 2^{-\alpha x}$, $\alpha > 0$ are widely used in non-correlated time decayed steaming data aggregation. It is easy to maintain simple sums and counts under such decay efficiently (21). However, in this section we will show that it is *not* possible to approximate C_τ^g with relative error guarantees using small space if m (the size of the universe) is large and g is exponential decay. This remains true for other classes of decay that are "faster" than exponential decay. We first present two natural approaches to approximate C_τ^g under an exponential decay function g, and analyze their space cost to show that each stores large amounts of information.

Algorithm I. Since tracking the sum under exponential decay can be performed efficiently using a single counter, we can just track the decayed correlated count for each distinct $v \in [m]$: $W_v^g = \sum_{e_i \in R | v_i = v} g(t - t_i)$, then $C_\tau^g = \sum_{v \geq \tau} W_v^g$. To ensure an good estimate for C_τ^g, each W_v^g must be tracked with sufficiently many bits of precision. One approach is that for each distinct $v \in [m]$ we maintain the timestamps of the last $\lceil \frac{1}{\alpha} \log_2 \frac{1}{\varepsilon} \rceil$ elements of the substream $R_v = \{v_i \in R | v_i = v\}$. From these timestamps, one can approximate W_v^g with a ε relative error bound, and hence C_τ^g can be approximated with an ε relative error bound. Each timestamp is $O(\log n)$ bits, so the total space cost is $O\left(m(\log n)\lceil \frac{1}{\alpha} \log \frac{1}{\varepsilon} \rceil\right)$ bits. □

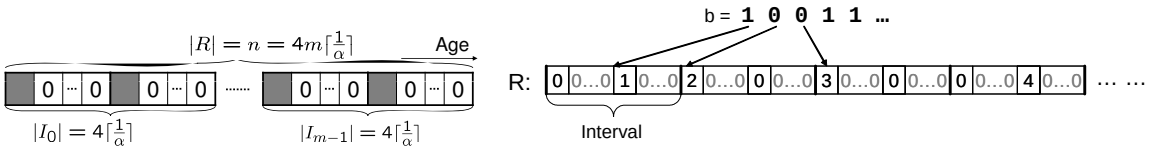
Algorithm II. The second algorithm tries to reduce the dependence on m by observing that for some close values of τ, the value of C_τ^g may be quite similar, so there is potential for "compression". As $g(x) = 2^{-\alpha x}$, $\alpha > 0$, we can write:

$$C_\tau^g = \sum_{v_i \geq \tau} 2^{\alpha(i-t)} = 2^{-\alpha t} \sum_{v_i \geq \tau} 2^{\alpha i}$$

where t is the query time. We reduce approximating C_τ^g with a relative error bound to a counting problem over an asynchronous stream with sliding window queries. We create a new stream R' in this model by treating each stream element as an item with timestamp set to its value v_i and with weight $2^{\alpha i}$. The query C_τ^g at time t can be interpreted as a sliding window query on the derived stream R' at time m with width $m - \tau + 1$. The answer to this query is $\sum_{v_i \geq \tau} 2^{\alpha i}$; by the above equation, scaling this down by $2^{\alpha t}$ approximates C_τ^g.

The derived stream R' can be summarized by sketches such as those in Chapter 2 or (14). These answer the sliding window query with relative error ε, implying relative error for C_τ^g. But the cost of these sketches applied here is $O(\frac{n}{\varepsilon} \log m)$ bits, linear in the stream length: in the reduction, the number of copies of each stream element increases exponentially, and the space cost of the sketches depends logarithmically on this quantity. □

Hardness of Exponential Decay. Algorithm I is a conceptually simple approach, which stores information for each possible value in the domain. Algorithm II uses summaries that are compact in their original setting, but when applied to the DCC problem, their space must increase to give an accurate answer for any τ. The core reason for the high space cost of both algorithms is the fact that as τ varies



(a) Setting the Intervals over a Stream (b) Mapping from binary string b to intervals

Figure 4.2 Creating a stream for the lower bound proof using $p = 1$

between 0 and m, the value of C_τ^g can vary over an exponentially large range, and a large data structure is required to track so many different values. This is made precise by the next theorem, which shows that the space cost of Algorithm I is close to optimal. We go on to provide a small space sketch with a weakened guarantee in Section 4.4.4, by limiting the range of values of C_τ^g for which an accurate answer is required.

Theorem 4.4.2. *For an exponential decay function $g(x) = 2^{-\alpha x}$, $\alpha > 0$ and $\varepsilon \leq 1/2$, any algorithm (one-pass or multi-pass, deterministic or randomized) that provides $\widehat{C}_\tau^g$ over a stream of length $n = \Theta(m)$, such that $|\widehat{C}_\tau^g - C_\tau^g| < \varepsilon C_\tau^g$ for any τ given at query time must store $\Omega(m \log \frac{n}{m})$ bits, where m is the universe size.*

Proof. The proof uses a reduction from the INDEX problem in two-party communication complexity (53). In the INDEX problem, Alice holds a binary string b of length N, and the second holds an index $i \in [N]$. Alice is allowed to send a single message to the second, who must then output the value of $b[i]$ (the ith bit of string b). Since no communication is allowed from Bob to Alice, the size of the message must be $\Omega(N)$ bits, even allowing the protocol a constant probability of failure (53).

We show that a small space streaming data structure to approximate DCC under exponential decay would allow a low communication complexity protocol for INDEX. Given a binary string b of length $N = mp$, we construct an instance of a stream, $R(b)$. Here m is the size of the domain of the stream values, and $p \geq 1$ is an integer parameter set later. The string b is divided into m partitions $\mathscr{P}_0, \mathscr{P}_1, \ldots, \mathscr{P}_{m-1}$, where $\mathscr{P}_i$ has bits $b[ip], b[ip+1], \ldots, b[(i+1)p-1]$.

Let $\ell = 2\lceil 1/\alpha \rceil$. The stream $R(b)$ has $n = m2^p\ell$ elements. The n positions in $R(b)$ are divided into m intervals, $I_0, I_1, \ldots, I_{m-1}$, each of length $2^p\ell$, as shown in Figure 4.2(a) for the case $p = 1$; the more recent elements of the stream belong to the lower numbered interval. Each interval I_j is further divided into 2^p segments, each with ℓ elements. Each element is a tuple (v, i) where v is the value and

i is the timestamp. The stream is synchronous, and the timestamps are consecutively increasing. The segments in I_j are numbered from 0 to $2^p - 1$, with the more recent segments in the stream getting smaller numbers. The value of every element of $R(b)$ is set to 0 except for m elements, one in each interval. The length p bit string in partition $\mathscr{P}_j$ in b is interpreted as an integer in the range $[0, 2^p - 1]$. In interval I_j, the segment numbered $\mathscr{P}_j$ is selected and the value of its most recent element is set to $j+1$, as shown in Figure 4.2(b) for the case $p = 1$.

Given a sketch that can approximate C_τ^g over $R(b)$ using ξ bits, we show a protocol for the INDEX problem with communication complexity ξ bits. Alice computes a sketch of $R(b)$, which can be used to approximate C_τ^g, and sends it to Bob. Bob, given an index i, computes $b[i]$ from the sketch, as follows. Let $\tau = \lfloor i/p \rfloor$. Note that bit $b[i]$ lies in partition $\mathscr{P}_\tau$ in b. Bob recovers the complete integer $\mathscr{P}_\tau$, by using the sketch to distinguish between different assignments to the bit string $\mathscr{P}_\tau$.

Consider two different assignments to the string $\mathscr{P}_\tau$, representing two integers p_1 and p_2. Without losing the generality, let $p_1 > p_2$ (p_1 and p_2 cannot be the same). Let the value of C_τ^g for these two assignments be C_1 and C_2. Suppose that the sketch of $R(b)$ provides an answer for C_τ^g which has a relative error of 1/2 or less. Let $\widehat{C}_1$ and $\widehat{C}_2$ denotes the estimates returned for C_1, C_2 respectively. In Lemma 4.4.1, we show that if $p_1 > p_2$, then $C_1 < C_2$ and $\widehat{C}_1 < \widehat{C}_2$. Thus, all the estimates for C_τ^g over different assignments for $\mathscr{P}_\tau$ are in a total order. So, by using an estimate of C_τ^g, we can distinguish between different assignments to $\mathscr{P}_\tau$, and hence recover all of $\mathscr{P}_\tau$, and solve the INDEX problem. Thus the size of the sketch of $R(b)$ must be at least $\Omega(N) = \Omega(mp)$ bits.

The stream length is $n = m2^p/\alpha$, so the lower bound on the sketch size is $N = \Omega(m\log(n/m))$, for a constant α. Since the communication lower bound allows randomization, the space lower bound also holds for randomized stream algorithms. Since we did not assume that Alice processed the stream in one pass, this space lower bound holds even if the stream is allowed to be processed in multiple passes. □

Let p_1 and p_2, C_1 and C_2, and $\widehat{C}_1$ and $\widehat{C}_2$ be defined as in the proof of Theorem 4.4.2.

Lemma 4.4.1. *If $p_1 > p_2$, then $C_1 < C_2$ and $\widehat{C}_1 < \widehat{C}_2$.*

Proof. Since the sketch provides estimates that are within a relative error of $1/2$, we have:

$$\frac{C_1}{2} \leq \widehat{C}_1 \leq \frac{3C_1}{2} \tag{4.7}$$

$$\frac{C_2}{2} \leq \widehat{C}_2 \leq \frac{3C_2}{2} \tag{4.8}$$

Let c denote the current time.

$$C_\tau^g = \sum_{\{(v,i) \in R(b) | v \geq \tau\}} g(c-i)$$

For integer $j, 0 \leq j \leq m-1$, let the contribution of interval j to C_τ^g be defined as:

$$C(j) = \sum_{\{(v,i) \in I_j | v \geq \tau\}} g(c-i)$$

Also, since the two bit strings differ in the values assigned to $\mathscr{P}_\tau$, the corresponding streams differ in interval I_τ. Let $C_1(\tau)$ and $C_2(\tau)$ respectively denote the value of $C(\tau)$ for the two inputs. We note that $C_1(\tau)$ and $C_2(\tau)$ differ by a factor of at least 4, since they both contain an element with the same value in I_τ, but in different positions, so that the decayed weights differ by a factor of at least 4. Since $p_1 > p_2$, according to the stream construction in the proof of Theorem 4.4.2, we have

$$C_1(\tau) \leq \frac{C_2(\tau)}{4} \tag{4.9}$$

In Lemma 4.4.2, we show that C_τ^g is dominated by the term $C(\tau)$. More precisely:

$$C(\tau) < C_\tau^g < \frac{4}{3} C(\tau) \tag{4.10}$$

Combining Inequalities 4.9 and 4.10, we get

$$C_1 < \frac{4}{3} C_1(\tau) \leq \frac{4}{3}\frac{1}{4} C_2(\tau) = \frac{1}{3} C_2(\tau) < \frac{1}{3} C_2 < C_2$$

Combining Inequalities 4.7, 4.8, and 4.9, we get:

$$\widehat{C}_1 \leq \frac{3}{2}C_1 < \frac{3}{2}\frac{4}{3}C_1(\tau) = 2C_1(\tau) \leq \frac{1}{2}C_2(\tau) < \frac{1}{2}C_2 \leq \widehat{C}_2$$

□

Lemma 4.4.2.

$$C(\tau) < C^g_\tau < \frac{4}{3}C(\tau)$$

Proof. Note that $C^g_\tau = \sum_{j=0}^{m-1} C(j)$. For every $0 \leq j < \tau$, we note that I_j does not have any tuples (v,i) with $v \geq \tau$. Thus, the contribution of such tuples to C^g_τ is 0, and so:

$$C^g_\tau = \sum_{j=\tau}^{m-1} C(j)$$

Also, for $j > \tau$, the contribution of $C(j)$ to C^g_τ is non-zero, since for $j > 0$, I_j has tuples (v,i) with $v \geq \tau$. Thus, $C(\tau) < C^g_\tau$, which proves one part of the Lemma.

Next, we note that for any integer ζ, no matter what the contents of $I_{\tau+\zeta}$ is,

$$C(\tau+\zeta) \leq \frac{C(\tau)}{4^\zeta}$$

The reason is as follows. Note that $C(\tau)$ is non-zero, since there is at one element in I_τ with value greater than or equal to τ. Further, there is exactly one non-zero element in each interval $I_{\tau+\zeta}$. Since the difference in timestamps between the non-zero element in I_τ and $I_{\tau+\zeta}$ is at least $\zeta \cdot \ell$, we have:

$$C(\tau+\zeta) \leq C(\tau) \cdot g(\ell\zeta) = C(\tau) \cdot 2^{-\alpha\ell\zeta} \leq \frac{C(\tau)}{4^\zeta}$$

where the last step follows from the definition of ℓ.

$$C^g_\tau = \sum_{j=\tau}^{m-1} C(j) = \sum_{\zeta=0}^{m-1-\tau} C(\tau+\zeta) \leq \sum_{\zeta=0}^{m-1-\tau} \frac{C(\tau)}{4^\zeta} < \sum_{\zeta=0}^{\infty} \frac{C(\tau)}{4^\zeta} = \frac{4}{3}C(\tau)$$

□

4.4.3 Super-exponential Decay

The result in Theorem 4.4.2 also holds for the super-exponential decay functions, defined in Section 1.4.1.

Theorem 4.4.3. *Consider a stream of length $n = \Theta(m)$, and a super-exponential decay function g. Any algorithm (one-pass or multi-pass, deterministic or randomized) that provides $\widehat{C}_\tau^g$, an estimate of C_τ^g, such that $|\widehat{C}_\tau^g - C_\tau^g| < \varepsilon C_\tau^g$ for any τ given at query time must store $\Omega\left(m \log \frac{n}{m}\right)$ bits, where m is the universe size.*

Proof. The proof is nearly identical to the one for Theorem 4.4.2, having the same structure and using the same reduction to the INDEX problem.

Again, Alice has a bit string b of length mp, which is divided into m partitions: $\mathscr{P}_0, \mathscr{P}_1, \ldots, \mathscr{P}_{m-1}$, where $\mathscr{P}_i$ has bits $b[ip], b[ip+1], \ldots, b[(i+1)p-1]$. Based on the bit string b, Alice creates a stream $R(b)$ of length $n = 2^p m\ell + c$, where $\ell = \lceil \log_\sigma 4 \rceil$ and c is the constant in the definition of super-exponential decay in Section 1.4.1, as follows. Each element is a (v,t) pair and elements are received in the order of their timestamps $0, 1, \ldots, 2^p m\ell + c - 1$, i.e., the stream $R(b)$ is strictly synchronous. The c elements with largest timestamps in $R(b)$ are assigned with value 0. Let $R'(b)$ denote the other elements in $R(b)$. For stream $R'(b)$, Alice assigns the values in the same way as she did for $R(b)$ in the proof for Theorem 4.4.2: (1) $R'(b)$ is divided into m intervals, $I_0, I_1, \ldots, I_{m-1}$, each of length $2^p\ell$; (2) Each interval I_j is further divided into 2^p segments, each with ℓ elements. The segments in I_j are numbered from 0 to $2^p - 1$, with the more recent segments in the stream getting smaller numbers; (3) The value of every element of $R'(b)$ is set to 0 except for m elements, one in each interval. The length p bit string in partition $\mathscr{P}_j$ in b is interpreted as an integer in the range $[0, 2^p - 1]$. In interval I_j, the segment numbered $\mathscr{P}_j$ is selected and the value of its most recent element is set to $j+1$.

Alice process the stream $R(b)$ and sends the sketch to Bob. Given the index i, Bob sets $\tau = \lceil i/p \rceil$ and queries the sketch for C_τ^g. Since the ages of the elements in $R'(b)$ are at least c, by the definition of super-exponential decay, any two neighboring elements in $R'(b)$ have their weights differ by a factor of at least σ. Thus, the two most recent elements in any two neighboring segments differ in their weights by a factor of at least 4, since we set $\ell = \lceil \log_\sigma 4 \rceil$. Further, since the c most recent elements in $R(b)$ will not have any contribution in C_τ^g, for any $\tau > 0$, because they are all assigned with value 0, we now have

the same argument between the bit string b and stream $R'(b)$ as we did for the bit string b and stream $R(b)$ in the proof for Theorem 4.4.2: by using $\widehat{C}_\tau^g$, the estimate of C_τ^g returned by the sketch, Bob can reveal the value of $b[i]$. So the space cost for processing stream $R(b)$ of length $n = 2^p m\ell + c$ is at least mp bits. By replace p, we get the space lower bound of $\Omega(m\log(n/m))$ bits, by constants of c and σ

Since the communication lower bound allows randomization, the space lower bound also holds for randomized stream algorithms. Since we did not assume that Alice processed the stream in one pass, this space lower bound holds even if the stream is allowed to be processed in multiple passes.

□

4.4.4 Finite (Super) Exponential Decay

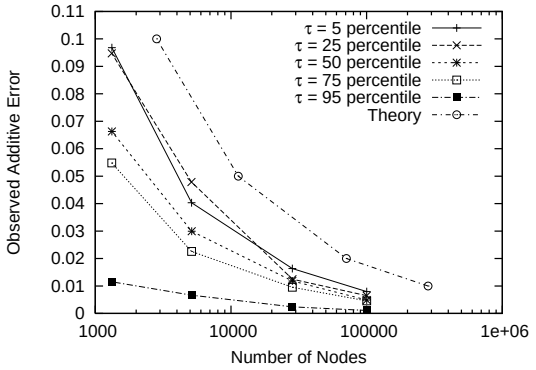
As noted above, the lower bound proof relies on distinguishing a sequence of exponentially decreasing possible values of the DCC. In practical situations, it often suffices to return an answer of zero when the true answer is less than some specified bound μ. This creates a "finite" version of exponential decay.

Definition 4.4.1. *A decay function g is a finite exponential decay function with threshold μ, $0 < \mu < 1$, if: (1) $g(x) = 2^{-\alpha x}$, $\alpha > 0$, if $x \leq \frac{1}{\alpha}\log_2 \frac{1}{\mu}$ (which implies $g(x) \geq \mu$); (2) $g(x) = 0$, otherwise.*

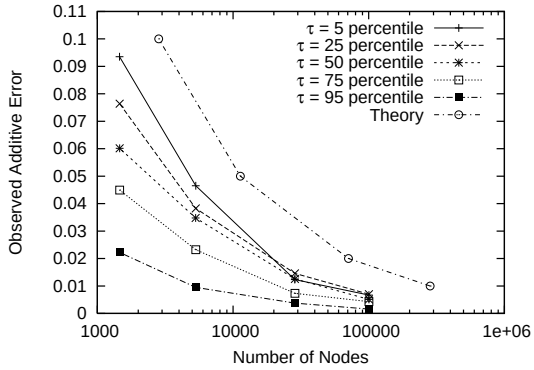
Since finite exponential decay is a finite decay, the space lower bound in Theorem 4.4.1 implies space of $\Omega((1/\alpha)\log(1/\mu))$ bits is necessary to approximate C_τ^g. A simple algorithm for C_τ^g simply stores all the stream elements with non-zero decayed weights. The space is $O((1/\alpha)\cdot\log m\cdot\log(1/\mu))$ bits, which is (nearly) optimal (treating $\log m$ as a small constant). This approach extends to the finite versions of super-exponential decay.

4.4.5 Sub-exponential decay

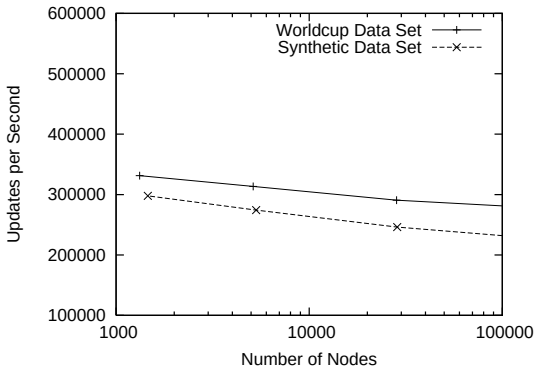
For any decay function $g(x)$, where $\lim_{x\to\infty} g(x) = 0$, we can always find $2^p m$ positions (timestamps) in the stream: $0 \leq x_1 < x_2 < \ldots < x_{2^p m}$, such that for every i, $1 < i \leq 2^p m$, we have $g(t - x_{i-1})/g(t - x_i) \leq 1/4$. Thus, it is natural to analyze what happens when we apply the construction from the lower bound in Theorem 4.4.2 to streams under such functions. Certainly, the same style of argument constructs a stream that forces a large data structure. If we fix some m and set $p = 1$, the



(a) Accuracy for world cup data set



(b) Accuracy for synthetic data set



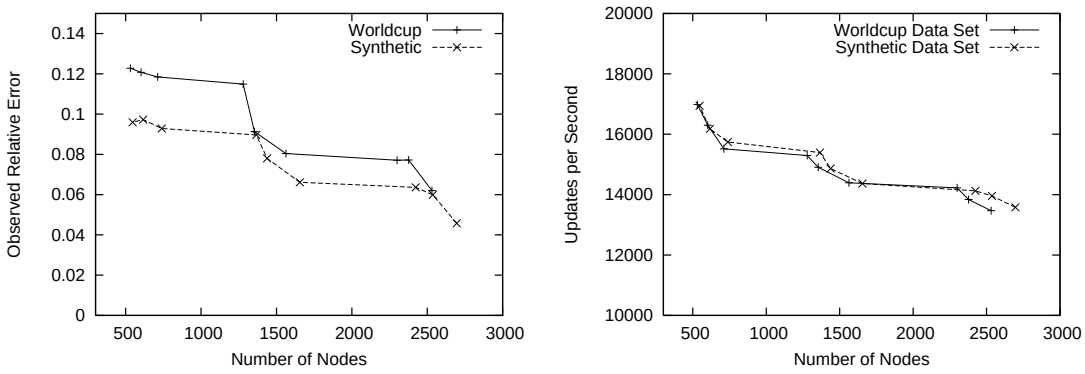
(c) Throughput versus space on both data sets

Figure 4.3 Throughput and accuracy with sliding window decay, additive error.

stream has to be truly enormous to imply a large space lower bound: e.g., for the polynomial decay function $g(x) = (x+1)^{-a}$, $a > 0$, we need $n = \Theta(2^{m/a})$ to force $\Omega(m)$ space. This is in agreement with the upper bounds in Section 4.3.2 which gave algorithms that depend logarithmically on n: for such truly huge values of n, this leads to a requirement of $\log 2^{m/a} = \Omega(m)$, so there is no contradiction.

4.5 Experiments

We present results from an experimental evaluation of the algorithms on two data sets. The first was web traffic logs from the 1998 World Cup on June 19th (the 'worldcup' data set) from `http://ita.ee.lbl.gov/`. Each stream element was a tuple (v,w,t), where v was the client id , w the packet size (modulo 100, simply to have initial weights bounded within a range), and t the timestamp, of the original web traffic logs, respectively. The dataset had 33695769 elements. The second



(a) Relative Error versus Space (b) Throughput versus Space

Figure 4.4 Performance of Relative Error Algorithm, with Polynomial Decay.

was a synthetically generated data set (the 'synthetic' data set). The size of the synthetic data is the same as the worldcup data set. Here, the timestamp of an element is a random number chosen uniformly from the range $[1, \max_t]$ where $\max_t = 898293600$ is the maximum timestamp in the world cup data set. The value v is chosen uniformly from the range $[1, \max_v]$, where $\max_v = 1823218$ is the maximum value in the worldcup data set. The weight is chosen similarly, i.e. uniformly from the range $[1, \max_w]$ where $\max_w = 99$ is the maximum weight in the world cup data.

We implemented our algorithms using C++/STL and all experiments were performed on a SUSE Linux Laptop with 1GB memory. Both input streams were asynchronous, and elements do not arrive in timestamp order.

Additive Error. We implemented the algorithm for additive error (Section 4.3.1) using the sketch in (30) as the basis. Note that the sketch in (30) provides the additional property of duplicate insensitivity, i.e., a re-insertion of the data into the sketch does not change the state of the sketch. Since our stream model does not have duplicates, our implementation of the sketch in (30) does not need to support duplicates detection and therefore improves the time efficiency in the stream processing.

Since a query for S_τ^g where g is not a sliding window decay can be reduced to queries for sliding window decay based DCS (21), we conducted experiments with the correlated sum S_τ^g where g is the sliding decay function. The window size is $4.5 \cdot 10^7$ for the synthetic data and 3600 for the worldcup data. We tried a range of values of the threshold τ, from the 5 percent quantile (5th percentile) of the values of stream elements to the 95 percent quantile. We analyzed the accuracy of the estimates

returned by the sketch, for a given space budget.

Figures 4.3(a) and 4.3(b) show the observed additive error as a function of the space used by the algorithm for different values of τ. The space cost is measured in the number of nodes, where each node is the space required to store a single stream element (v,w,t), which takes a constant number of bytes. This cost can be compared to the naive method which stores all input elements (nearly 34 million nodes). The observed error is usually significantly smaller than the guarantee provided by theory. The theoretical guarantee holds irrespective of the value of τ or the window size. Note that the additive error decreased as the square root of the space cost, as expected. Figure 4.3(c) shows the throughput, which is defined as the number of stream elements processed per second, as a function of the space used. From the results, the trend is for the throughput to decrease slowly as the space increases. Across a wide range of values for the space, the throughput is between 250K and 350K updates per second.

Relative Error. We implemented WBMH and the sketch designed in Chapter 2 as the bucket sketch embedded in WBMH. We performed similar experiments to test our algorithms for relative error, based on the polynomial decay function $g(x) = 1/(x+1)^{1.5}$, a non-exponential converging decay. The thresholds are the same as in the additive error algorithm. The results are shown in Figure 4.4. In general, the space cost for a given error for polynomial decay was much smaller than the algorithm for sliding windows (Figure 4.4(a)). This greater space efficiency comes at some cost: we have to fix the decay function *a priori*—the additive error result allows the decay function to be specified at query time. The throughput for the relative error algorithm is also appreciably lower than the additive error algorithm (Figure 4.4(b)), by over an order of magnitude. This is partly due to the greater time complexity of the relative error algorithm caused by the periodic bucket merging operations which access every node in the merged buckets, and partly because our implementation is not fully tuned.

4.6 Concluding Remarks

Our results shed light on the problem of computing correlated sums over time-decayed streams. The upper bounds are quite strong, since they apply to asynchronous streams with arbitrary timestamps. It is also possible to extend these results to a distributed streaming model, since the summarizing data structures used can naturally be computed over distributed data, and merged together to give a

summary of the union of the streams. The lower bounds are similarly strong, since they apply to the most restricted model, for computing DCC where there is exactly one arrival per time unit.

The correlated sum is at the heart of many correlated aggregates, but there are other natural correlated computations to consider which do not follow immediately from DCS. Some we expect to be hard in general: correlated maximum $\max_{v_i > \tau} w_i g(t - t_i)$ has a linear space lower bound under finite decay functions, since this lower bound follows from the uncorrelated case. Other analysis tasks seem feasible but challenging: for example, to output a good set of cluster centers for those points with $v_i > \tau$, weighted by $w_i g(t - t_i)$. It will be of interest to understand exactly which such correlated aggregations are possible in a streaming setting.

CHAPTER 5. Forward Decay: A Practical Decay Model for Stream Systems

As we have shown in the last few chapters, temporal data analysis in 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 (except for the correlated data aggregation), 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 time decay, defined in Definition 1.4.1, 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 chapter, 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 a variety of backward decay functions, such as exponential decay and landmark windows. 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.

5.1 Introduction

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 (76) and heartbeats (50) are used to avoid query blocking. A number of general purpose prototype streaming systems have been created, such as Stream (73), Aurora (69) and TelegraphCQ (17); the current state-of-the-art deployed streaming systems (including GS (33) and Streambase (74)) 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 (33). 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 (23). 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 (35; 42; 25), 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 (surveyed in more detail in Section 5.7) 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* (Definition 1.4.1). That is, the weight of an item is computed based on its age, measuring *back* from the current time. 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.

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 5.2 we describe decay models and existing backward decay definitions, then in Section 5.3 we introduce our model of forward decay and study its properties. We show how to compute aggregates under forward decay in Section 5.4, and how to draw samples in Section 5.5. Implementation issues are discussed in Section 5.6, related work in Section 5.7, and our experimental study is described in Section 5.8.

5.2 Decay functions

We consider a stream of items (v_i, t_i), projected on the dimensions of the *value* and *timestamp* from the stream R, defined in Chapter 1.3. We first give a more general definition of decay function which can abstract the backward decay model defined in Definition 1.4.1 and the forward decay model.

Definition 5.2.1. *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 $\mathscr{W}(i,t)$, or just $\mathscr{W}(i)$ when t is implicit. We define a function $\mathscr{W}(i,t)$ to be a* decay function *if it satisfies the following properties:*
1. $\mathscr{W}(i,t) = 1$ when $t_i = t$ and $0 \leq \mathscr{W}(i,t) \leq 1$ for all $t \geq t_i$.
2. $\mathscr{W}$ is monotone non-increasing as time increases: $t' \geq t \Rightarrow \mathscr{W}(i,t') \leq \mathscr{W}(i,t)$.

5.2.1 Backward Decay Functions

Prior work on time decay, including those presented in previous chapters, has focused on decay functions of a particular form: where the weight of an item can be written as a function of its initial weight and its *age*, x, where the age at time $t > t_i$ is simply $x = t - t_i$. 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 5.2.2. *A backward decay function is defined by a positive monotone non-increasing function $g()$ so that the weight of the ith item at time t is given by*

$$\mathscr{W}(i,t) = \frac{g(t-t_i)}{g(t-t)} = \frac{g(t-t_i)}{g(0)}$$

The denominator in the expression normalizes the weight, so that it obeys condition 1 of Definition 5.2.1. Some examples of the most popular decay functions are generated by picking g to be of a certain form, such as *no decay, sliding window decay, exponential decay* and *polynomial decay*, defined in Chapter 1.4, by adding an appropriate denominator. It is easy to verify that all the above functions satisfy the requirements for decay functions (Definition 5.2.1).

There has been significant study of how to compute a variety of simple and complex aggregates under decay functions (21; 57; 25) (especially the special case of sliding window (7; 35; 54)). 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 5.7.

5.3 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 5.2.1. 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 y. In order to normalize values given that the function value increases with time, we typically need to include a normalizing factor in terms of $y(t)$, the function of the current time. More formally,

Definition 5.3.1. *Given a positive monotone non-decreasing function y, 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*

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

This definition ensures that when $t = t_i$ the weight is 1 (condition 1 of Definition 5.2.1). As t increases, this weight never increases (due to the monotonicity of y) and remains in the range $[0,1]$. Observe that scaling y by a constant has no effect on the value of the decayed weight.

Example 5.3.1. *Consider the stream of (t_i, v_i) pairs*

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

Let the landmark time $L = 100$, and set $y(n) = n^2$. 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 5.1. □

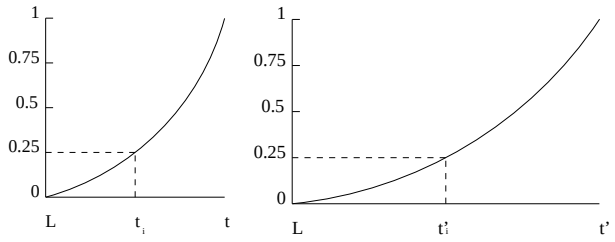
As with backward decay, the most natural choices of functions y fall into similar classes:

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

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

5.3.1 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 $g(a) =$



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

Figure 5.1 Relative decay property for forward decay on $y(n) = n^2$

$\exp(-\alpha(a))$, its decayed weight is $\mathscr{W}i,t) = \exp(-\alpha(t-t_i))$. Under forward decay, its decayed weight under the function $y(n) = \exp(\alpha(n))$ is

$$\frac{y(t_i - L)}{y(t - L)} = \frac{\exp(\alpha(t_i - L))}{\exp(\alpha(t - L))} = \exp(\alpha t_i - \alpha L - \alpha t + \alpha L) = \exp(-\alpha(t - t_i)) = \mathscr{W}i,t)$$

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 5.5).

5.3.2 Polynomial Decay

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

Definition 5.3.2. *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 5.3.1. *Forward decay based on a monomial function $y(n) = n^\beta$ satisfies the relative decay property.*

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

$$\mathscr{W}i,t) = \frac{y(t_i - L)}{y(t-L)} = \frac{y(\gamma(t-L))}{y(t-L)} = \frac{(\gamma(t-L))^\beta}{(t-L)^\beta} = \gamma^\beta$$

□

This is illustrated in Figure 5.1 with $y(n) = n^2$: at time t, in Figure 5.1(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 5.1(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.

5.3.3 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 (43). 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.

5.4 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 y and a landmark time L.

5.4.1 Count, Sum and Average

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

Definition 5.4.1 (Count, Sum and Average)**.** *The decayed count, C, is the sum of decayed weights of stream items*

$$C = \sum_{i=1}^{n} \frac{y(t_i - L)}{y(t - L)}$$

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

$$S = \sum_{i=1}^{n} \frac{v_i \cdot y(t_i - L)}{y(t - L)}$$

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

$$A = S/C = \left(\sum_i y(t_i - L)v_i\right) / \left(\sum_i y(t_i - L)\right)$$

Example 5.4.1. *Take the same example stream given in example 5.3.1. 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 $S = \frac{1}{y(t-L)}(\sum_i y(t_i - L)v_i)$. This can be computed by maintaining the value of $\sum_i y(t_i - L)v_i$, and scaling by the value of $y(t-L)$ 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, $V = \sum_i y(t_i - L)v_i^2/C - A^2$. 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 $y(t_i - L)$. The final result is found by scaling the sum by $y(t-L)$ at query time t. Thus:

Theorem 5.4.1. *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 $1+\varepsilon$ relative error for any backward decay function, but requiring a blow up in space by an $O(\frac{1}{\varepsilon}\log n)$ factor.

5.4.2 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 5.4.2 (Min and Max)**.** *The decayed minimum value MIN is defined as*

$$MIN = \min\left(\frac{v_i \cdot y(t_i - L)}{y(t-L)}\right) = \frac{1}{y(t-L)} \min_i (v_i \cdot y(t_i - L))$$

and the decayed maximum value MAX is defined as

$$MAX = \max\left(\frac{v_i \cdot y(t_i - L)}{y(t-L)}\right) = \frac{1}{y(t-L)} \max_i (v_i \cdot y(t_i - L))$$

Observe that in both cases it suffices to compute the smallest (greatest) value of $y(t_i - L)v_i$ seen so far. For MAX, when a new (t_i, v_i) pair is observed, compute the corresponding value of $y(t_i - L)v_i$, 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.

5.4.3 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 5.7).

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

Definition 5.4.3 (Heavy hitters under forward decay)**.** *For each item in the input, v, its decayed count is given by $d_v = \sum_{v_i=v} y(t_i - L)/y(t-L)$. Given a threshold value ϕ, the ϕ heavy-hitters are all items v satisfying $d_v \geq \phi C$.*

Example 5.4.2. *Consider the example stream given in Example 5.3.1. 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 5.4.1. 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 5.4.2. *Given an error bound ε, we find all items with $d_v \geq \phi C$, and report no items with $d_v < (\phi - \varepsilon)C$ under the forward decay model using space $O(1/\varepsilon)$ counters, and processing each update in time $O(\log 1/\varepsilon)$.*

Proof. Observe that we can rewrite the requirement as

$$d_v y(t-L) \geq \phi C y(t-L)$$
$$\text{or equivalently } \sum_{v_i=v} y(t_i - L) \geq \phi \sum_i y(t_i - L).$$

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 $y(t_i - L)$. Importantly, these weights do not change over time.

We can use the SpaceSaving algorithm proposed by Metwally *et al.* (59). As analyzed in (24), this algorithm naturally extends to weighted updates. We omit full details of the proof for brevity; the proof in (24) 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 5.4.4 (Quantiles under forward decay)**.** *For each item v, its decayed rank is computed as* $r_v = \sum_{v_i \leq v} y(t_i - L)/y(t - L)$. *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 5.4.3. *Given an error bound* ε*, we find decayed* ϕ*-quantiles under forward decay using space* $O(\frac{1}{\varepsilon} \log U)$ *counters, and processing each update in time* $O(\log \log U)$.

Proof. Similarly to heavy hitters, we can factor out the $y(t - L)$ term, so that we reduce the problem to find the smallest item i such that $\sum_{v_i \leq v} y(t_i - L) \geq \phi \sum_i y(t_i - L)$. This is a weighted quantiles problem defined over the (static) weights $y(t_i - L)$. We can now make use of solutions to weighted quantiles problems. The q-digest data structure (71; 24) 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 (46; 48)): factor out the $y(t - L)$ term and track the input using weights $y(t_i - L)$. We suppress further examples that fit this pattern for brevity.

5.4.4 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 5.4.5 (Count Distinct under forward decay)**.** *The distinct count D of a set of items under forward decay is*

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

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

Theorem 5.4.4. *Given a desired error bound ε, we can approximate D under the forward decay model within relative error $(1 \pm \varepsilon)$ using space $\tilde{O}(1/\varepsilon^2)$.*

Proof. We write distinct count under forward decay as

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

and so focus our effort on estimating the quantity $\sum_v \max_{v_i=v} y(t_i - L)$, which does not depend on the query time t. As before, $y(t_i - L)$ can be computed on arrival of the item, and does not vary with time. So we can write the weight of item i as $\mathscr{W}(i,t) = w_i = y(t_i - L)$, and the desired quantity is $\sum_v \max_{v_i=v} w_i$. This now corresponds exactly to the "dominance norm" defined in (26). The most efficient method to approximate this quantity is due to Pavan and Tirthapura (67), which generalizes techniques for counting the number of distinct items. Applied to our problem, the time cost is $\tilde{O}(1/\varepsilon^2)$

(with $\tilde{O}$ notation suppressing polynomial factors in $\log n$ and $\log \varepsilon$). Each update takes time $\tilde{O}(1)$ time. The result is correct up to relative error $1 \pm \varepsilon$ with high probability. □

5.5 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.

5.5.1 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

$$\frac{\mathscr{W}(i,t)}{\sum_{i=1}^{n} \mathscr{W}(i,t)} = \frac{y(t_i - L)}{\sum_{i=1}^{n} y(t_i - L)}$$

Theorem 5.5.1. *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 $W_i = \sum_{j=1}^{i} y(t_j - L)$ 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 $y(t_i - L)/W_i$. The probability that the ith item is chosen as the final sample is given by

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

□

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 (77), the procedure can be accelerated by using an appropriate random distribution to determine the total weight of subsequent items to skip over.

5.5.2 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 (65). 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 $y(t_i - L)$ 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 (37) 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 (37) 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 $y(t-L)$ in forward decay and this does not affect the sampling probability for each element, we can set the weight of each tuple $w_i = y(t_i - L)$, and obtain a sample according to the weights in the forward decay model.

Priority Sampling. Priority sampling due to Alon *et al.* (4) 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 5.5.2. *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.

5.5.3 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 5.5.2.1. *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 5.5.2, and the fact shown in Section 5.3.1 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.

5.6 Implementing Forward Decay

5.6.1 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 $y(t_i - L)$, and only scale by $y(t - L)$ at query time. But while $y(t_i - L)/y(t - L)$ is guaranteed to lie between zero and one, the intermediate values of $y(t_i - L)$ 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 y values (scaled sums), they can be rescaled relative to a new landmark. That is, by the analysis of exponential decay in Section 5.3.1, 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.

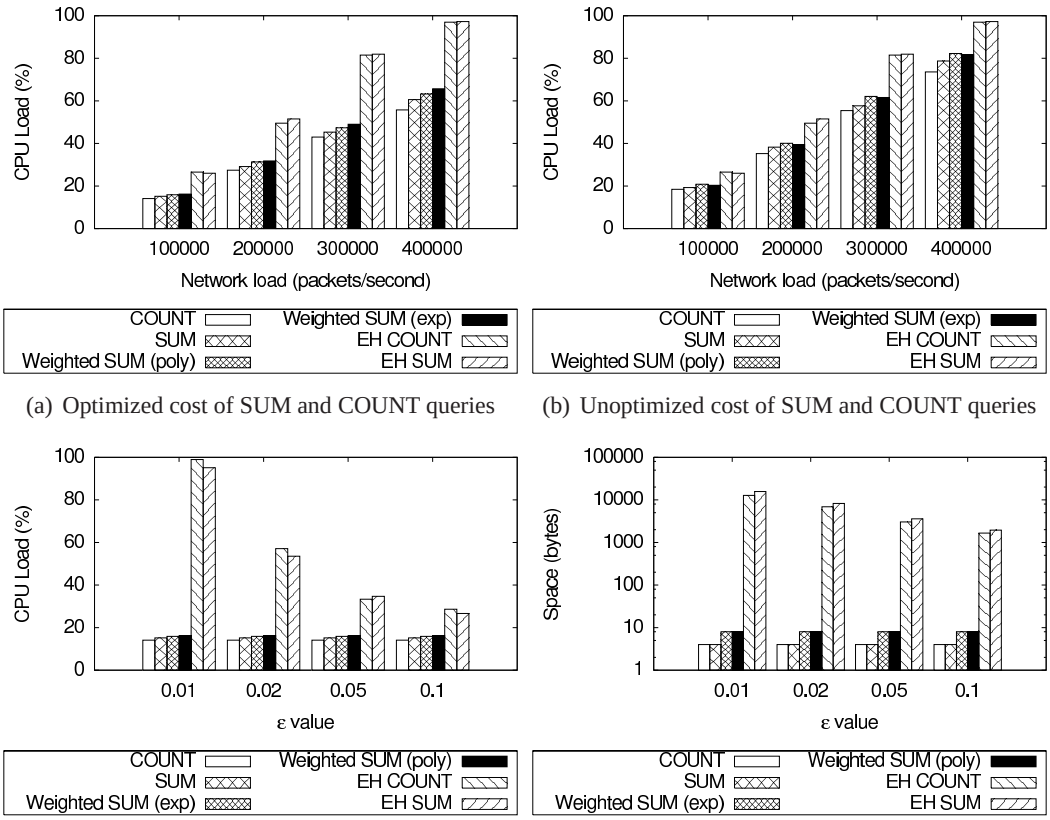
5.6.2 Out-of-order and Distributed arrivals

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 (14; 25). 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.

5.7 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.



(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

Figure 5.2 Experiments on Count queries under time decay

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 (43) and references therein). Various models have been proposed for the semantics of sliding windows. The Aurora system (15) 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.* (55) 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" (34).

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) (35) and Deterministic Waves (42) answer these queries on a window of size N with relative error ε by keeping a careful arrangement of $O(\frac{1}{\varepsilon}\log \varepsilon N)$ 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 $\frac{1}{\varepsilon}$ 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 $\log \frac{1}{\varepsilon} \log N$ factor larger than the unwindowed approximate algorithms (7). Lee and Ting (54) reduce the space for frequent items for a fixed size window to $O(\frac{1}{\varepsilon})$, 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 (14), sampling (30) and quantiles and heavy hitters (25). 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 (9).

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 (24), 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 (21), with cost $O(\frac{1}{\varepsilon}\log N)$. This was extended to sampling and aggregate computation (30; 25), with similar blow-ups of $\text{poly}(\frac{1}{\varepsilon}, \log N)$ 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.

5.8 Experimental Evaluation

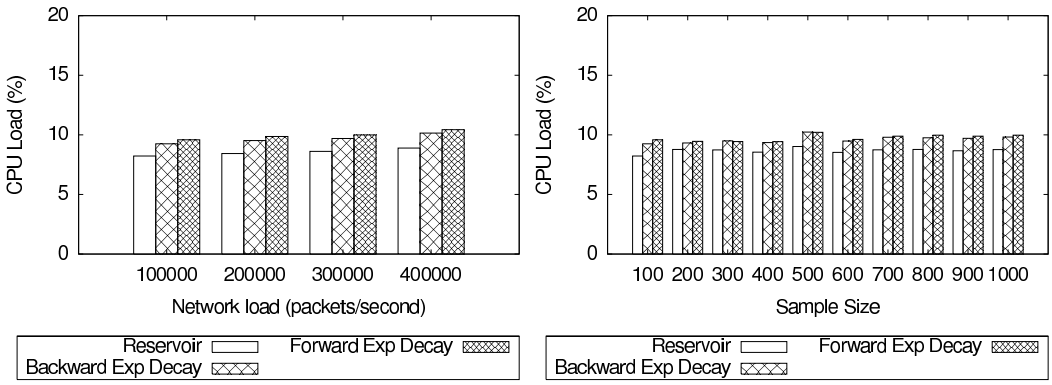
In this section we present the results of experimental evaluation of several aggregate and sampling streaming algorithms under forward and backward decay models.

Experimental Set-Up and Environment. All the experiments were done in the context of the GS streaming database (33). 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) (35), with variations for both sum and count. This makes for an interesting comparison, since, following the analysis of Cohen and Strauss (21), 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 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 (77), and compared the cost of this to priority sampling being supplied with exponentially increasing weights (4) 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 5.4.3[1]. Here, we compared to a method for answering sliding window heavy hitter queries (25). 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

[1]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

Figure 5.3 Experiments on Sampling Queries under time decay

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.

5.8.1 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 5.2. Figure 5.2(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 $\varepsilon = 0.1$) 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 5.2(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.

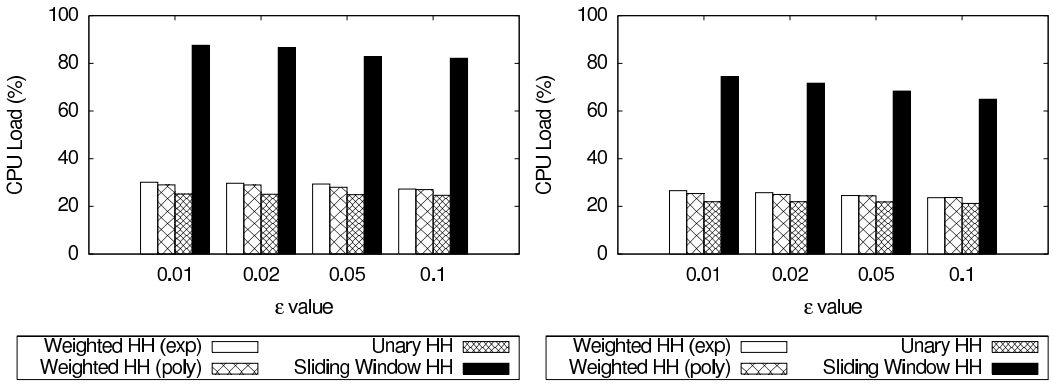
This benefit becomes more pronounced as we vary the accuracy parameter ε of the exponential histograms. Recall that exponential histograms give an answer that is approximate to within relative error $1+\varepsilon$, while the other queries are computed exactly. For the same queries as before, we decreased ε down to 0.01, while the stream data rate was set to 100,000 packets/second (Figure 5.2(c)). The throughput of undecayed and forward decayed aggregates does not alter, since they do not depend on ε. At $\varepsilon = 0.01$, 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 5.2(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 5.3. The sampling techniques are all implemented as UDAFs in C code, which are then called by GSQL queries as

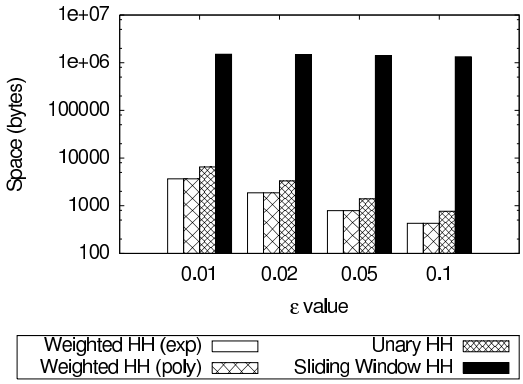
```
select tb, PRISAMP(srcIP, exp(time % 60))
from TCP
group by time/60 as tb
```

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 (exponen-

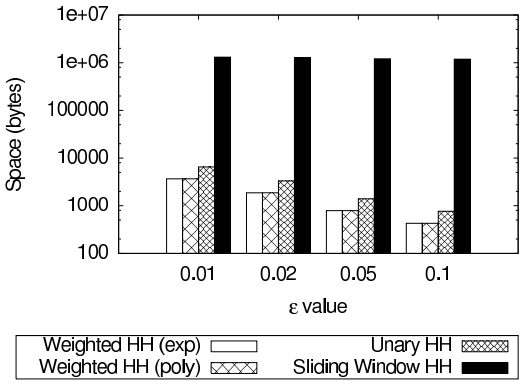


(a) CPU load as ε varies (TCP data)

(b) CPU load as ε varies (UDP data)



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



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

Figure 5.4 Experiments on Heavy Hitter queries under time decay

tial) 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 5.3(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 5.3(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,

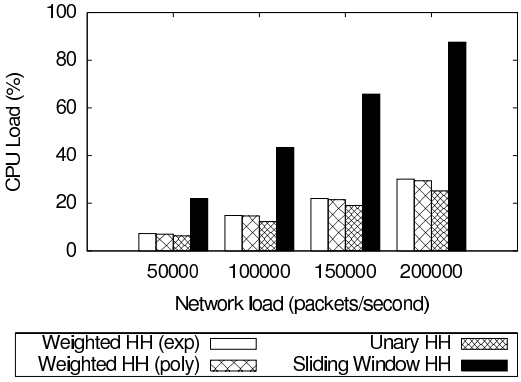


Figure 5.5 Experiments on HH performance as stream rate varies

plus some small additional values such as 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 5.5 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.

This is further highlighted in Figures 5.4(a) and 5.4(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 $\varepsilon = 0.01$, 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 $1/\varepsilon$ (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 5.4(b) and 5.4(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).

5.9 Concluding Remarks

In this chapter, we have proposed a new class of time decay for streaming systems, based on a forward view of the decay. 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 (36), Hadoop (47) and Sawzall (68).

CHAPTER 6. Conclusion

In this thesis, we proposed the concept of asynchronous data streams. We showed asynchronous data stream is a more natural model for the streaming data transmitted through distributed systems than the previous synchronous data stream model, and it is therefore a robuster model for distributed data stream monitoring.

We focused on the time-decayed data aggregation over asynchronous data streams. We showed that previous work on synchronous data stream cannot be trivially extended for the asynchronous data stream. We proposed the first time and space efficient sketches for summarizing multiple asynchronous data streams over timestamp sliding windows. The sketch is further improved so that it can be used for general purposed network streaming data monitoring in a communication-efficient way.

Our techniques for asynchronous data stream processing were further explored in its usage for correlated data aggregation over data streams. We not only closed the open problem of sliding window based correlated data aggregation, but also did the first comprehensive study on the correlated data aggregation for asynchronous data streams under any arbitrary time decay functions. We proposed time and space efficient algorithms for the easy cases, and showed large space lower bounds for the hard cases.

We proposed forward-decay, a new time decay model for down weighting old elements in the stream. We showed that forward-decay captures a variety of time decay functions in the usual backward decay model. We showed forward decay significantly simplifies the data aggregation and sampling over data streams by providing much simpler and more efficient algorithms.

We conclude this thesis with the following future directions.

6.1 Future Work

In general, all the problems that have been studied using the synchronous data stream model should be re-investigated when the asynchrony in the data stream becomes a matter. Although in some scenarios a solution for the synchronous stream case can be used for the asynchronous stream case by paying extra cost in the time and space, it is of research interest whether this extra cost can be waived. In other scenarios, novel techniques become a must. We list a few of such examples here.

Population variance. Maintaining a good estimate for the population variance over sliding windows of a synchronous stream is well studied (10; 80) and optimal solutions exist, while the same problem under the asynchronous stream model still remains open. Histogram techniques used in the synchronous stream case highly relies on the sequential order in the data arrival, and it is not known how to extend it for the asynchronous stream. Similarly, it is not known how to use sampling technique to maintain a probabilistically accurate estimate for the variance over sliding windows of an asynchronous stream.

Tighter bound for basic counting. Basic Counting over a sliding window is a fundamental problem in stream processing (35). The known best space lower bound of $\Omega(\log^2 w/\varepsilon)$ bits, where w is the window size, is optimal for both deterministic and randomized algorithms in the synchronous stream case. The current best space upper bound for asynchronous stream is $O((\log^3 w/\varepsilon))$ bits for deterministic algorithms (30) and $O(\log^2 w/\varepsilon^2)$ bits for randomized algorithms (79). Closing the gap between the lower and upper bounds is an open problem.

Tighter bound for heavy hitters The best space upper bound for maintaining heavy hitters over sliding windows is $O(1/\varepsilon)$ words (54) for the synchronous stream case. The idea is to use the Misra and Gries algorithm (60) but replace the simple counter with a sketch counter (called λ-counter in (54)), which approximately tracks the number of each distinct value in the window. An observation is that the sketch counter only needs to be of additive error guarantee. However, λ-counter does not work in the asynchronous stream setting. The sketch counters designed in (30; 79) for asynchronous streams are also over-killed since they have a relative error guarantee, and thus are too complex and inefficient in space. Finding a more space efficient sketch counter for basic counting over sliding windows in asynchronous streams may lead to more efficient and simpler algorithms for heavy hitters.

Semi-asynchronous streams. So far our asynchronous data stream model does not assume an upper bound in the transmission delay of any data element. In reality, if the data element is delayed at any intermediate device for longer than a preset threshold, it will be discarded. Therefore, the real data stream observed by the processor has an upper bound in the latency of its data arrival. We call such data streams, where the data arrival latency is bounded, as semi-asynchronous streams. By taking advantage this latency bound, we may improve the performance of existing algorithms which were designed for asynchronous stream processing.

BIBLIOGRAPHY

[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 *the Proceedings of the ACM International Conference on Management of Data*, pages 882–884, 2005.

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

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

[4] N. Alon, N. Duffield, C. Lund, and M. Thorup. Estimating arbitrary subset sums with few probes. In *the Proceedings of the ACM Symposium on Principles of Database Systems*, pages 317–325, 2005.

[5] 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.

[6] R. Ananthakrishna, A. Das, J. Gehrke, F. Korn, S. Muthukrishnan, and D. Srivastava. Efficient approximation of correlated sums on data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):569–572, 2003.

[7] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *the Proceedings of the ACM Symposium on Principles of Database Systems*, pages 286–296, 2004.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *the Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–16, 2002.

[9] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 633–634, 2002.

[10] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *the Proceedings of the ACM Symposium on Principles of Database Systems*, pages 234 – 243, 2003.

[11] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, 1970.

[12] V. Braverman and R. Ostrovsky. Smooth histograms for sliding windows. In *the Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 283–293, 2007.

[13] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[14] C. Busch and S. Tirthapura. A deterministic algorithm for summarizing asynchronous streams over a sliding window. In *the Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 465–476, Aachen, Germany, 2007.

[15] 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 *the Proceedings of the International Conference on Very Large Data Bases*, pages 215–226, 2002.

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

[17] 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 *the Proceedings of the ACM International Conference on Management of Data*, pages 668–668, 2003.

[18] D. Chatziantoniou and K. A. Ross. Querying multiple features of groups in relational databases. In *the Proceedings of the ACM International Conference on Very Large Data Bases*, pages 295 – 306, Bombay, India, 1996.

[19] Y. Chen, H. V. Leong, M. Xu, J. Cao, K. C. C. Chan, and A. T. S. Chan. In-network data processing for wireless sensor networks. In *the Proceedings of the International Conference on Mobile Data Management*, pages 26–29, 2006.

[20] E. Cohen and H. Kaplan. Spatially-decaying aggregation over a network. *Journal of Computer and System Sciences*, 73(3):265–288, 2007.

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

[22] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *the Proceedings of the International Conference on Data Engineering*, pages 449–460, 2004.

[23] G. Cormode, F. Korn, S. Muthukrishnan, T. Johnson, O. Spatscheck, and D. Srivastava. Holistic udafs at streaming speeds. In *the Proceedings of the ACM International Conference on Management of Data*, pages 35–46, 2004.

[24] G. Cormode, F. Korn, and S. Tirthapura. Exponentially decayed aggregates on data streams. In *the Proceedings of the International Conference on Data Engineering*, pages 1379–1381, 2008.

[25] G. Cormode, F. Korn, and S. Tirthapura. Time-decaying aggregates in out-of-order streams. In *the Proceedings of the ACM Symposium on Principles of Database Systems*, pages 89–98, Vancouver, Canada, 2008.

[26] G. Cormode and S. Muthukrishnan. Estimating dominance norms of multiple data streams. In *the Proceedings of the European Symposium on Algorithms*, pages 148–160, 2003.

[27] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *the Proceedings of the ACM Symposium on Principles of Database Systems*, pages 271–282, 2005.

[28] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu. Forward decay: A practical time decay model for streaming systems. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 138–149, 2009.

[29] G. Cormode, S. Tirthapura, and B. Xu. Time-decayed correlated aggregates over data streams. *To appear in Statistical Analysis and Data Mining*.

[30] G. Cormode, S. Tirthapura, and B. Xu. Time-decaying sketches for sensor data aggregation. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 215–224, Portland, Oregon, USA, 2007.

[31] G. Cormode, S. Tirthapura, and B. Xu. Time-decayed correlated aggregates over data streams. In *the Proceedings of SIAM International Conference on Data Mining*, pages 269–280, 2009.

[32] G. Cormode, S. Tirthapura, and B. Xu. Time-decaying sketches for robust aggregation of sensor data. *SIAM Journal on Computing*, 39(4):1309–1339, 2009.

[33] C. Cranor, L. Gao, T. Johnson, and O. Spatscheck. Gigascope: high performance network monitoring with an sql interface. In *the Proceedings of the ACM International Conference on Management of Data*, pages 623–623, 2002.

[34] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *the Proceedings of the ACM International Conference on Management of Data*, pages 647–651, 2003.

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

[36] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[37] P. S. Efraimidis and P. G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

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

[39] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[40] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *the Proceedings of the ACM International Conference on Management of Data*, pages 13–24, Santa Barbara, CA, USA, 2001.

[41] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *the Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 281–291, 2001.

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

[43] L. Golab. *Sliding window query processing over data streams*. PhD thesis, Computer Science, University of Waterloo, 2006.

[44] M. Greenwald and S. Khanna. Space efficient online computation of quantile summaries. In *the Proceedings of the ACM International Conference on Management of Data*, pages 58–66, 2001.

[45] S. Guha, D. Gunopulos, and N. Koudas. Correlating synchronous and asynchronous data streams. In *the Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*, pages 529–534, 2003.

[46] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *the Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 359–366, November 2000.

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

[48] P. Indyk. Better algorithms for high-dimensional proximity problems via asymmetric embeddings. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 539–545, 2003.

[49] P. Indyk and D. Woodruff. Tight lower bounds for the distinct elements problem. In *the Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 283–283, 2003.

[50] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in gigascope. In *the Proceedings of the ACM International Conference on Management of Data*, pages 1079–1088, 2005.

[51] 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*, pages 8–13, 2005.

[52] 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.

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

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

[55] 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.

[56] 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.

[57] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *the Proceedings of the International Conference on Data Engineering*, pages 767–778, 2005.

[58] G. Manku, S. Rajagopalan, and B. Lindsley. Approximate medians and other quantiles in one pass and with limited memory. In *the Proceedings of the ACM International Conference on Management of Data*, pages 426–435, June 1998.

[59] A. Metwally, D. Agrawal, and E. A. Abbadi. Efficient computation of frequent and top-k elements in data streams. In *the Proceedings of the International Conference on Database Theory*, pages 398–412, 2005.

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

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

[62] S. Muthukrishnan. Data streams: Algorithms and applications. Technical report, Rutgers University, Piscataway, NJ, 2003.

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

[64] S. Nath, P. B. Gibbons, S. Seshan, and Z. Anderson. Synposis diffusion for robust aggregation in sensor networks. In *the Proceedings of the International Conference on Embedded Networked Sensor Systems*, pages 250–262, 2004.

[65] F. Olken. *Random Sampling from Databases*. PhD thesis, Computer Science, U. C. Berkeley, 1993.

[66] B. Patt-Shamir. A note on efficient aggregate queries in sensor networks. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 283–289, 2004.

[67] 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.

[68] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[69] S. Z. Sbz, S. Zdonik, M. Stonebraker, M. Cherniack, U. C. Eintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26, 2003.

[70] J. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

[71] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *the Proceedings of the International Conference on Embedded Networked Sensor Systems*, pages 239–249, Baltimore, Maryland, USA, 2004.

[72] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *the Proceedings of the ACM Symposium on Principles of Database Systems*, pages 263–274, 2004.

[73] Stanford stream data manager. `http://infolab.stanford.edu/stream`.

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

[75] S. Tirthapura, B. Xu, and C. Busch. Sketching asynchronous streams over a sliding window. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 82–91, 2006.

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

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

[78] 1998 WorldCup web requests:. `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`.

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

[80] L. Zhang and Y. Guan. Variance estimation over sliding windows. In *the Proceedings of the ACM Symposium on Principles of Database Systems*, 2007.