# Sketching Asynchronous Streams Over a Sliding Window

Srikanta Tirthapura
Dept. of ECE
Iowa State University
2215, Coover Hall
Ames, IA 50011, USA
snt@iastate.edu

Bojian Xu
Dept. of ECE
Iowa State University
2215, Coover Hall
Ames, IA 50011, USA
bojianxu@iastate.edu

Costas Busch
Dept. of Computer Science
Rensselaer Polytechnic Inst.
110 8th Street
Troy, NY 12180, USA
buschc@cs.rpi.edu

## ABSTRACT

We study the problem of maintaining sketches of recent elements of a data stream. Motivated by applications involving network data, we consider streams that are *asynchronous*, in which the observed order of data is not the same as the time order in which the data was generated. The notion of recent elements of a stream is modeled by the *sliding timestamp window*, which is the set of elements with timestamps that are close to the current time. We design algorithms for maintaining sketches of all elements within the sliding timestamp window that can give provably accurate estimates of two basic aggregates, the sum and the median, of a stream of numbers. The space taken by the sketches, the time needed for querying the sketch, and the time for inserting new elements into the sketch are all polylog with respect to the maximum window size and the values of the data items in the window. Our sketches can be easily combined in a lossless and compact way, making them useful for distributed computations over 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. Our notion of recency of elements is more general than that studied in previous work, and thus our sketches are more robust to network delays and asynchrony.

## Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed Systems; H.2.8 [**Database Applications**]: Data Mining

## General Terms

Algorithms, reliability, performance

The work of the authors was supported in part through NSF grants CNS 0520102 and CNS 0520009.

## Keywords

Data stream processing, aggregates, sliding windows, sketches of streams, union of streams, asynchronous streams, distributed streams

## 1. INTRODUCTION

Enormous quantities of data flow through today's computer networks everyday. Often, it is necessary to analyze this massive volume of data "on the fly" and compute aggregates and statistics, and detect trends and anomalies in traffic. The need for processing such data has led to the study of the *data stream* model of computation, where the data has to be processed in a single pass using workspace that is typically much smaller than the size of the stream.

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 [2, 6, 9, 4, 5, 13] on designing algorithms for maintaining aggregates over a *sliding window* of the most recent elements of a data stream. So far, all work on maintaining aggregates over a sliding window has assumed that 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 explain next.

**Asynchronous Streams.** In many real-life situations involving 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 order of arrival of stream elements is not necessarily the order in which the elements were generated.

For example, nodes in a sensor network generate observations, each tagged with a timestamp. When the observations are transmitted to an *aggregator* node (sometimes also referred to as a *sink*), the inherent network asynchrony and differing link delays may cause an observation with an earlier timestamp to reach the aggregator later than an observation with a more recent timestamp. We call such a data stream, where the order of receipt of elements is not necessarily the order of generation of the data, as an *asynchronous data stream*. Thus, 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 observation. The greater the timestamp of an observation, the more recent the data.

Asynchronous data streams are inevitable anytime two streams of observations, say streams $A$ 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 each individual streams $A$ or $B$ are not inherently asynchronous, i.e. elements within $A$ or within $B$ arrive in increasing order of timestamps, 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 aggregator may consistently observe elements with earlier timestamps from $B$ after elements with more recent timestamps from $A$.

All previous work on maintaining aggregates on a sliding window of a data stream have considered the case of synchronous arrivals, where it is assumed that the stream elements arrive in order of increasing timestamps. In this paper we present the first study of aggregate computation over a sliding window of a data stream under asynchronous arrivals.

We consider the following model for processing asynchronous data streams. In the centralized model, a single aggregator node $A$ receives a data stream $R = d_1, d_2, \ldots, d_n$ where $d_1$ is the observation that was received the earliest, and $d_n$ the observation that was received most recently. Each observation $d_i$ is a tuple $(v_i, t_i)$ where $v_i$ is the data and $t_i$ is the timestamp, which is tagged at the time the data was generated. Since we consider asynchronous arrivals, it is not necessary that the $t_i$s are in increasing order, i.e. it is possible that $i > j$ (so that $d_i$ is received by $A$ later than $d_j$) but $t_i < t_j$. Let $c$ denote the current time at any instant. The user will ask the aggregator queries of the following form: return an aggregate (say, the sum or the average) of all elements in the stream that have timestamps which are within the range $[c-w, c]$, where $w$ is the width of the "window" of timestamps. Since the window $[c-w, c]$ is constantly changing with the current time $c$, we refer to this range $[c-w, c]$ as the "sliding timestamp window". When the context is clear, we sometimes use the term "sliding timestamp window" to refer to all received items that have timestamps in the range $[c-w, c]$.

The challenge with maintaining aggregates over a sliding timestamp window is that the data within the window can be very large and it is usually infeasible to store all the elements in the window 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.

**Distributed Streams.** 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 (sink). 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 which the nodes are typically battery operated devices.

Unlike previous work [9, 8, 13] that considered the synchronous model on distributed streams, we consider aggregate computation over distributed streams under asynchronous arrivals. In our approach, we place aggregators in a hierarchical structure, such as a spanning tree. Sketches are transmitted up the tree from the leaves to the root, and are combined in a distributed way as they move up the tree. Finally the root node has the sketch of the union of all the streams, and can answer aggregate queries about the union of all streams (see Figure 7). The sketch of the union can be constructed on demand, whenever new queries are issued. Further, the sketches can be combined in a way that the relative error is small and also the size of the sketch of the union of two streams does not increase more than a fixed bound.

## 1.1 Contributions

First, we give algorithms for computing the sum and median of the sliding timestamp window of an asynchronous stream that is being observed by a single aggregator. We then consider the distributed case, where we give a procedure that combines the sketches produced by the aggregators, each of which is observing and sketching a local stream. In the discussion below, $R = (v_1, t_1), (v_2, t_2), \ldots, (v_n, t_n)$ is an asynchronous stream of positive integer, timestamp pairs. Let $c$ denote the current time, and $W$ a bound on the maximum window size.

**Sum of Positive Integers.** Our first sketching algorithm estimates the sum of all integers in stream $R$ which are within any recent timestamp window of size $w \le W$, i.e. $V_w = \sum_{\{(v,t)\in R | c-w \le t \le c\}} v$. 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 $(\epsilon, \delta)$-estimator to quantify the quality of answers returned by the algorithm.

Definition 1.1. *For parameters $0 < \epsilon < 1$ and $0 < \delta < 1$, an $(\epsilon, \delta)$-estimator for a number $Y$ is a random variable $X$ such that $\Pr[|X - Y| > \epsilon Y] < \delta$.*

Our algorithm has the following performance guarantees.

- For any $w \le W$ specified by the user at the time of the query, the sketch returns an $(\epsilon, \delta)$-estimator of $V_w$. 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 $O(\frac{1}{\epsilon^2}\log(1/\delta)\log V_{max}\log m)$, where $V_{max}$ is a known upper bound on $V_w$, and $m$ is an upper bound on the value of each data item $v$.
- The expected time to process an item $(v,t)$ is $O((\log v)(\log\log(1/\delta)+\log(1/\epsilon)))$.
- Time taken to process a query for the sum is $O(\log\log V_{max}+\frac{1}{\epsilon^2}\log(1/\delta))$.

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* in [5]. Our algorithm solves basic counting immediately by taking $v=1$ for every data item.

**Median of Elements.** 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 $(\epsilon,\delta)$-approximate median is defined as follows.

Definition 1.2. *An $(\epsilon,\delta)$-approximate median of a totally ordered set $S$ is a random variable $Z$ such that the rank of $Z$ in $S$ is between $(1/2-\epsilon)|S|$ and $(1/2+\epsilon)|S|$ with probability at least $1-\delta$.*

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 $(\epsilon,\delta)$-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 $O(\frac{1}{\epsilon^2}\log(1/\delta)\log m\log W)$ bits, where $m$ is the maximum value of an input observation.
- The expected time taken to process each item is $O(\log\log 1/\delta+\log 1/\epsilon)$.
- Time taken to process a query for the median is $O(\log\log W+\frac{1}{\epsilon^2}\log(1/\delta))$.

**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, and can be used for the composition of multiple sketches 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 $A$ (or $B$) consists of a series of random samples from the input stream $A$ (or $B$). 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 $(\epsilon,\delta)$-estimator, then the combined sketch can yield an $(\epsilon,\delta)$-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, and good accuracy guarantees.

## 1.2 Related Work

Datar *et al.* [5] considered basic counting over a sliding window of elements in a data stream under the synchronous arrival model. They presented an algorithm that is based on a novel 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 $\ell_p$ norms, to basic counting. For a sliding window size of maximum size $W$, and an $\epsilon$ relative error, the space taken by their algorithm for basic counting is $O(\frac{1}{\epsilon}\log^2 W)$, 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 $O(\frac{1}{\epsilon}\log W(\log W+\log m))$, and worst case time complexity of $O(\log W+\log m)$ where $m$ is an upper bound on the value of an item.

We now 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 many 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 more recent elements are present in smaller size buckets than older elements, and all bucket sizes are powers of two. In an asynchronous stream, however, the element that just arrived may have an earlier timestamp, and may fit into an "old" bucket, which could be of size greater than 1. Inserting this item into the "old" bucket causes the following problems: (1)The current size of the old bucket may no longer be a power of 2, and (2)It is not clear how to merge this bucket with neighboring buckets, should the size of the data structure exceed the desired maximum in the future. It seems that the exponential histogram is dependent on the elements arriving in the order of timestamps.

Later, Gibbons and Tirthapura [9] gave an algorithm for basic counting based on a data structure called the *wave* that used the same space as in [5], 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.

Arasu and Manku [2] 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.* [4] presented algorithms for maintaining the variance and $k$-medians of elements within a sliding window of a data stream. Feigenbaum *et al.* [6] considered the problem of maintaining the diameter of a set of points in the sliding window model.

Gibbons and Tirthapura [8] introduced a model of distributed computation over data. 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 [8], 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 [9], they considered estimation of functions over a sliding window on distributed streams. However, the algorithms in [9] were designed for the case of synchronous arrivals. Patt-Shamir [16] 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 [11] 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 [19] designed a *heartbeat* generation algorithm to support continuous queries in a Data Stream Management System(DSMS), 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. Their work does not consider maintenance of aggregates, as we do here.

Much other recent work on data stream algorithms has been surveyed in [3, 15]. To our knowledge, our work is the first to consider maintaining aggregates over a sliding window under asynchronous arrivals.

## 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 $v_i$s are positive integers and $t_i$s are the timestamps. Let $c$ denote the current time at the aggregator.

**Problem Description:** 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 $R_w$. $V = \sum_{\{(v,t)\in R_w\}} v$

### 2.1 Intuition

Our algorithm for the sum is based on *random sampling*. The high level idea is that 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.

The following intuition about random sampling guides our algorithm design. If random sampling is used in estimating the cardinality of a set of elements, in order to get a desired accuracy for the estimate, it is enough to sample the elements of the set such that the resulting sample is "large enough"; what is "large enough" depends only on the desired accuracy ($\epsilon$ and $\delta$), and not on the size of the set itself. The required size of the sample can be determined using Chernoff bounds. In getting random sampling to work for our estimation problem, we need the following ideas.

Firstly, in estimating the sum, different elements in the stream have to be treated with different weights, 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 important for guaranteeing the error bounds, and for further details on this we refer the reader to the formal description of the algorithm.

Secondly, 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. Our algorithm maintains not just one random sample, but many random samples, at probabilities $p = 1, \frac{1}{2}, \frac{1}{4}, \ldots$. (note that our treatment of elements of different weights makes the sampling probabilities higher for elements with greater value). In each sample, we maintain only the most recent elements selected into the sample, and discard the older elements. We show that when a query is asked, with high probability, one of these samples will provide a good estimate for the sum of all elements within the sliding timestamp window.

## 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 $\log V_{max}$, so the upper bound $V_{max}$ can be a very loose upper bound. For example, if the stream had no more than $f$ elements having the same timestamp, and an upper bound $m$ was known on each value $v$, then $fmW$ is a trivial upper bound on $V$.

Let $M = \log V_{max}$. 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 timestamp of the element that was most recently 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 $\alpha = \frac{12}{\epsilon^2} \ln\left(\frac{8}{\delta}\right)$.

The initialization step of the algorithm appears in Figure 1. Figure 2 shows how to update the sketch upon receiving an element, and Figure 3 shows how to use the sketch in answering a query for the sum.

```
For every $i = 0 \ldots M$
    1. $S_i \leftarrow \phi$ /* All samples initially empty */
    2. $t_i \leftarrow -1$ /* No items have been discarded yet*/
```

**Figure 1: Sketch for the Sum: Initialization**

## 2.3 Correctness Proof

Let $X$ denote the result of Algorithm 3 when a query is asked for the sum of elements within the sliding timestamp window $[c-w, c]$. We show that $X$ is an $(\epsilon, \delta)$-estimate of $V$, the sum of elements within the window.

For the purpose of the proof, it is convenient to define $T_i$, the set of all elements in $R_w$ that were ever inserted by the algorithm into $S_i$. Note that $T_i$ is not completely stored by the algorithm, but only the $\alpha$ most recent elements of $T_i$ are stored in $S_i$. More formally, we have the following definition.

DEFINITION 2.1. *For $i = 0, \ldots, M$, $T_i$ is the set constructed by the following probabilistic process. For each element $(v,t) \in R_w$, if $\frac{v}{2^i} \geq 1$, then insert $(\frac{v}{2^i}, t)$ into $T_i$. Let $\ell$ denote the smallest integer such that $v/2^\ell < 1$. Insert $(1,t)$ into $T_\ell$ with probability $v/2^\ell$. If $(1,t)$ was selected into level $\ell$, then generate a sequence of independent tosses of an unbiased coin, whose probability of heads in each toss is $1/2$. Insert $(1,t)$ into succeeding levels $\ell+1, \ell+2, \ldots, M$ as long as the coin comes up heads, and stop the insertion the first time the coin comes up tails.*

The following fact follows from the definition of $T_i$.

```
1. If $(t < c - W)$, then discard $d$ since it is outside the
   largest timestamp window, and a future query will
   never involve $d$. Return.
2. Let $\ell$ = smallest integer $i$, $0 \leq i \leq M$, such that
   $v/2^\ell < 1$
3. For each level $i = 0, \ldots, \ell - 1$
   (a) Insert $(v/2^i, t)$ into $S_i$
   (b) If $|S_i| > \alpha$, then discard element with lowest
       timestamp in $S_i$, say $t'$.
   (c) Update $t_i \leftarrow \max\{t_i, t'\}$
4. With probability $v/2^\ell$, insert $(1,t)$ into $S_\ell$
   (a) If $|S_\ell| > \alpha$, then discard element with lowest
       timestamp in $S_\ell$, say $t'$.
   (b) Update $t_\ell \leftarrow \max\{t_\ell, t'\}$.
5. Set $i \leftarrow \ell + 1$
6. While $(1,t)$ was inserted into $S_{i-1}$ and $i \leq M$,
   (a) Insert $(1,t)$ into $S_i$ with probability $1/2$.
   (b) If $|S_i| > \alpha$, then discard element with lowest
       timestamp in $S_i$, say $t'$.
   (c) Update $t_i \leftarrow \max\{t_i, t'\}$
   (d) $i \leftarrow i + 1$
```

**Figure 2: Sketch for the Sum: Processing an element $d = (v,t)$**

FACT 2.1. *For $i = 0 \ldots M$, if $|T_i| > \alpha$ then after observing stream $R$, $S_i$ contains $\alpha$ items with the most recent timestamps in $T_i$. If $|T_i| \leq \alpha$, then $S_i = T_i$.*

DEFINITION 2.2. *For each element $d = (v,t) \in R$, for each $i = 0, 1, \ldots, M$ define random variable $x_i(d)$ as follows. If $v \geq 2^i$ then $x_i(d) = v/2^i$. If $v < 2^i$ then $x_i(d) = 1$ with probability $v/2^i$ and $0$ with probability $1 - v/2^i$.*

LEMMA 2.1. *If $d = (v,t)$ then $E[x_i(d)] = v/2^i$*

PROOF. If $v \geq 2^i$ then $E[x_i(d)] = v/2^i$, since $x_i(d)$ is a constant. If $v < 2^i$, then, since $x_i(d)$ is a 0-1 random variable, $E[x_i(d)] = \Pr[x_i(d) = 1] = v/2^i$ □

DEFINITION 2.3. *For $i = 0, \ldots, M$, define $X_i = \sum_{(v',t) \in T_i} v'$.*

LEMMA 2.2. *For $i = 0, \ldots, M$, we have $E[X_i] = \frac{V}{2^i}$*

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

$$X_i = \sum_{(v',t) \in T_i} v' = \sum_{d=(v,t) \in R_w} x_i(d)$$

Using linearity of expectation and Lemma 2.1, we get

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

1. Let $\ell$ be the smallest integer $i$, $0 \le i \le M$ such that $t_i < c - w$
2. If $\ell$ exists then return $2^\ell \sum_{\{(v',t) \in S_\ell | t \ge c-w\}} v'$
3. If $\ell$ does not exist, then return /* Algorithm Fails */

**Figure 3: Sketch for the Sum: When an estimate is asked for the sum of elements in timestamp window $[c-w, c]$.**

□

Next, we need to show that the sample that is used by the algorithm returns a good estimate for the sum. The following definition captures the notion of whether or not different samples yield good estimates for $V$.

Definition 2.4. *For $i = 0, \ldots, M$, random variable $X_i$ is said to be "good" if $(1-\epsilon)V \le X_i 2^i \le (1+\epsilon)V$, and "bad" otherwise. Define event $B_i$ to be true if $X_i$ is bad, and false otherwise.*

Lemma 2.3. *If $|R_w| \le \alpha$, then the algorithm returns the exact answer for the sum.*

Proof. In this case, all elements of $R_w$ will be stored in $S_0$, which will be used by the algorithm to return the exact sum of elements. □

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

Definition 2.5. *Let $\ell^\star \ge 0$ be an integer such that $E[X_{\ell^\star}] \le \alpha/2$ and $E[X_{\ell^\star}] > \alpha/4$.*

Lemma 2.4. *Level $\ell^\star$ is uniquely defined and exists for every input stream $R$.*

Proof. From Lemma 2.2, we have $E[X_i] = V/2^i$. Since $V > \alpha$, $E[X_0] > \alpha$. By the definition of $M = \log V_{max}$, it must be true that $V \le 2^M$ for any input stream $R$, so that $E[X_M] \le 1$. 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 $E[X_{\ell^\star}] \le \alpha/2$ and $E[X_{\ell^\star}] > \alpha/4$. □

For the next lemmas, we use Hoeffding bounds. We use the version of the bounds from Schmidt, Siegel and Srinivasan [18] (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 2.5. **Hoeffding's Bound (restated from [18]):**
*(1) If $0 < \delta < 1$, then $\Pr[Y > \mu(1+\delta)] \le e^{-\mu\delta^2/3}$.*
*(2) If $\delta \ge 1$, then $\Pr[Y > \mu(1+\delta)] \le e^{-\mu\delta/3}$.*
*(3) If $0 < \delta < 1$, then $\Pr[Y < \mu(1-\delta)] \le e^{-\mu\delta^2/2}$.*

The next lemma helps in the proof of Lemma 2.7.

Lemma 2.6. *If $0 < a < \frac{1}{2}$ and $k \ge 0$, then $a^{(2^k)} \le \frac{a}{2^k}$*

Proof. It is clear by induction that $2^k - 1 \ge k$. Since $0 < a < \frac{1}{2}$, we can further get $a^{(2^k-1)} \le a^k < \left(\frac{1}{2}\right)^k$. Therefore, $a^{(2^k)} < \frac{a}{2^k}$. □

The next lemma shows that it is highly unlikely that $B_i$ is true for any $i$ such that $0 \le i \le \ell^\star$.

Lemma 2.7. *For integer $\ell$ such that $0 \le \ell \le \ell^\star$,*

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

Proof.

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

From Definition 2.2, 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, $X_\ell$ 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

$$E[X_\ell] = c + \mu_Y \tag{1}$$

By the definition of $\ell^\star$, $E[X_{\ell^\star}] > \alpha/4$. Since $E[X_i] = \frac{V}{2^i}$ (from Lemma 2.2). Using Equation 1, we get the following inequality that will be used in further proofs.

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

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

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

Where $\delta' = \frac{\epsilon(c+\mu_Y)}{\mu_Y}$.

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

**Case I:** $\delta' < 1$. Using Lemma 2.5 and the fact $(c+\mu_Y)/\mu_Y \ge 1$, we have

$$\Pr[Y > \mu_Y(1+\delta')] \le e^{-\mu_Y\delta'^2/3} = e^{-\frac{\epsilon^2(c+\mu_Y)^2}{3\mu_Y}} \le e^{-\epsilon^2(c+\mu_Y)/3}$$

$$e^{-\epsilon^2(c+\mu_Y)/3} < e^{-\epsilon^2(2^{\ell^\star-\ell}(\alpha/4))/3} < \left(\frac{\delta}{8}\right)^{(2^{\ell^\star-\ell})} \le \frac{\delta/8}{2^{\ell^\star-\ell}}$$

Where we have used $\alpha = 12\frac{\ln 8/\delta}{\epsilon^2}$ and $\delta < 1$, Equation 2 and Lemma 2.6.

**Case II:** $\delta' \ge 1$. Using Lemma 2.5, we have:

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

where we have used $\alpha = 12\frac{\ln 8/\delta}{\epsilon^2}$ and $\delta < 1$, Equation 2 and Lemma 2.6.

From Case I and Case II, we have

$$\Pr[X_\ell < (1+\epsilon)E[X_\ell]] < \frac{\delta/8}{2^{\ell^\star-\ell}} \qquad (3)$$

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

$$\begin{aligned}\Pr[X_\ell < (1-\epsilon)E[X_\ell]] &= \Pr[c+Y < (1-\epsilon)(c+\mu_Y)] \\ &= \Pr[Y < \mu_Y(1-\delta')]\end{aligned}$$

.
Where $\delta' = \frac{\epsilon(c+\mu_Y)}{\mu_Y}$
Using Lemma 2.5 and the fact $\frac{\mu_Y+c}{\mu_Y} \geq 1$,

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

Using Equation 2 and Lemma 2.6,

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

Thus, we have

$$\Pr[X_\ell < (1-\epsilon)E[X_\ell]] < \frac{\delta/8}{2^{\ell^\star-\ell}} \qquad (4)$$

Combining Equations 3 and 4, for $0 \leq \ell \leq \ell^\star$, we get

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

□

Lemma 2.8.

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

Proof. By definition of $B_i$, $\Pr[B_i] = \Pr[2^iX_i \notin (1-\epsilon,1+\epsilon)V] = Pr[X_i \notin (1-\epsilon,1+\epsilon)E[X_i]]$
Using Lemma 2.7,

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

□

The following precisely characterizes the behavior of the algorithm.

Fact 2.2. *When answering a query for the sum of all relevant items, the algorithm uses level $\ell$ (i.e. sample $S_\ell$), where $\ell$ is the smallest integer such that $|T_\ell| \leq \alpha$. If no such integer $\ell$ exists, the algorithm fails.*

Suppose the algorithm used level $\ell$ to answer the query for the sum.

Lemma 2.9.

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

Proof. If $\ell > \ell^\star$, it follows from Fact 2.2 that $|T_{\ell^\star}| > \alpha$, else the algorithm would have stopped at level $\ell \leq \ell^\star$. This further implies that $X_{\ell^\star} > \alpha$, since for each $(v',t) \in T_{\ell^\star}$, $v' \geq 1$. Thus we have

$$\Pr[\ell > \ell^\star] < \Pr[X^\star_\ell > \alpha] \qquad (5)$$

As in the proof of Lemma 2.7, we denote $X^\star_\ell = c + Y$, where $c$ is a constant and $Y$ is the sum of independent 0-1 random variables. Let $\mu_Y = E[Y]$. Since $E[X^\star_\ell] \leq \alpha/2$, 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 2.5,

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

Where $\delta' = \frac{c+\mu_Y}{\mu_Y} > 1$.
Since, $E[X_{\ell^\star}] = c + \mu_Y > \alpha/4$, we have

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

□

Theorem 2.1. *The result of the algorithm, $X$, is an $(\epsilon,\delta)$-estimate for $V$, the sum of all elements in the timestamp window $[c-w,c]$.*

Proof. Let $\ell$ denote the level used by the algorithm for answering a query for $V$. Let $f$ denote the probability that the algorithm fails to return an estimate that is within an $\epsilon$ 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 2.8 and 2.9. □

## 2.4 Complexity

Lemma 2.10. **Space Complexity:** *The total space taken by the sketch for the sum is $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\log V_{max}\log m)$ where $V_{max}$ is an upper bound on the value of the sum $V$, $m$ is an upper bound on the value of each data item $v$, $\epsilon$ is the desired relative error, and $\delta$ is the desired upper bound on the failure probability.*

Proof. The algorithm maintains $M = \log V_{max}$ samples, each of which has up to $\alpha = 12\frac{\ln 8/\delta}{\epsilon^2}$ elements. Each element in the sample is a pair $(v',t)$. Each $v'$ is a value that is either 1, or is of the form $v/2^i$ where $v$ is a positive integer in $[1,m]$

and $i \in [0, \lceil \log m \rceil]$. Thus, $v'$ can be stored using $2 \log m$ bits. We assume that the timestamp $t$ can also be stored in $O(\log m)$ space. Thus, each element $(v, t)$ can be stored in $O(\log m)$ space. The product of the number of samples, the number of elements per sample, and the space per element yields the above space complexity. $\square$

LEMMA 2.11. **Time Complexity:** *The expected time taken for handling an element $(v, t)$ is $O((\log v)(\log \log 1/\delta + \log 1/\epsilon))$, and the worst case time is $O((\log V_{max})(\log \log 1/\delta + \log 1/\epsilon))$. The time taken to answer a query for the sum is $O(\log \log V_{max} + \frac{\log 1/\delta}{\epsilon^2})$, where $V_{max}, \epsilon, \delta$ are as described in Lemma 2.10.*

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 $\alpha = 12\frac{\ln 8/\delta}{\epsilon^2}$.

The time taken to handle an element $d = (v, t)$ consists of two parts. The first part is the deterministic insertion of the element into $\log v$ samples $S_i$ for all $i$ such that $2^i \leq v$. Each insertion into a sample may cause a overflow, which can be handled by a delete-min on the heap. Thus, the time for the deterministic insertion into samples is $O(\log v \log \alpha)$.

The second part is the sampling of $d$ into later levels. Let $\ell$ be the smallest integer such that $v/2^\ell < 1$. Element $d$ is inserted into level $\ell$ with probability $v/2^\ell$, and if it is inserted into level $i \geq \ell$, it is inserted into level $i + 1$ with probability $1/2$, and this process continues until the element fails to be inserted into a level, or till level $M$ is reached. Due to this random process, the expected number of levels that $d$ is inserted into is no more than 2, and the worst case number of levels is no more than $M$. Thus the expected time taken for the second part is $O(\log \alpha)$, and the worst case time for the second part is $O(M \log \alpha)$.

Summing the two parts, the expected total time to handle a new arrival $(v, t)$ is $O(\log \alpha \log v)$, and the worst case time is $O(\log \alpha \log V_{max})$ (in the worst case, the element is inserted into every level of the data structure). Of course, if element $d$ is outside the window, then it takes constant time to discard it.

The time taken to answer a query again consists of two parts. The first part is to determine the appropriate sample for answering this query. This may take $O(M)$ time in the worst case if we searched the levels one by one. However, if binary search is used to determine the appropriate level, this can be done in $O(\log M) = O(\log \log V_{max})$ worst case time. The second part of answering a query for the sum consists of actually computing the sum of the relevant elements in the appropriate sample. Since a sample contains no more than $\alpha$ elements, this cost is $O(\alpha)$. The total cost for answering a query for the sum is this $O(\log \log V_{max} + \alpha)$. $\square$

## 3. COMPUTING THE MEDIAN

It is well known that computing the exact median of a data stream requires linear space; pretty much every element in the stream has to be stored in the workspace of the aggregator. The linear space lower bound holds even for the computation of the median of elements within a sliding window of a data stream. Thus, we consider the computation of an $(\epsilon, \delta)$-approximate median (defined in Section 1) of all elements within the sliding timestamp window.

More precisely, consider a data stream $R$ whose elements are arriving asynchronously. Given a maximum window size $W$, we design a sketch that can return an $(\epsilon, \delta)$-approximate median of all elements in $R$ whose timestamps are in the range $[c - w, c]$, where $w \leq W$ is the window size that is supplied by the user at the time of the query.

Just as in the case of estimating the sum, random sampling is a useful tool for computing approximate medians, and is the basis of many median and quantile finding algorithms for data streams [14, 10]. Roughly speaking, the median of a random sample of a stream, where the stream is sampled at a "large enough" probability, is an approximate median of all elements in the stream. What is a "large enough" probability depends on the size of the set on which the median is being computed, and the accuracy desired. Since the window size $w$ is known only at query time, there is no single sampling probability for the element 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. In contrast with the algorithm for the sum, 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. The formal statement of the algorithm for the median appears in Figures 4, 5 and 6. We note that the same idea in the algorithm for the median computation can also be used in computing other quantiles of the data stream under asynchronous arrivals.

The size of each sample is $\alpha = 96\frac{\ln(8/\delta)}{\epsilon^2}$. Let $R_w = \{(v, t) \in R | c - w \leq t \leq c\}$ denote the set of elements in the current timestamp window of width $w$.

```
For every $i = 0 \dots M$
    1. $S_i \leftarrow \phi$ /* All samples initially empty */
    2. $t_i \leftarrow -1$ /* No items have been discarded yet*/
```

**Figure 4: Sketch for the Median: Initialization**

THEOREM 3.1. *Algorithm 6 returns an $(\epsilon, \delta)$-approximate median of all elements within $R_w$.*

The proof is omitted due to space constraints.

LEMMA 3.1. **Space Complexity:** *The total space taken by the sketch for the median is $O(\frac{\log 1/\delta}{\epsilon^2} \log m \log W)$ bits, where $m$ is the maximum value of an input observation, $W$ the maximum window size, $\epsilon$ the relative error, and $\delta$ the failure probability.*

LEMMA 3.2. **Time Complexity:** *The expected time taken for handling an element $(v, t)$ is $O(\log \log 1/\delta + \log 1/\epsilon)$. The time taken to answer a query for the median is $O(\log \log W + \frac{\log 1/\delta}{\epsilon^2})$*

The proofs of the above two lemmas have been omitted due to space constraints.

```
1. If ($t < c - W$), then discard $d$ since it is outside the
   largest timestamp window, and a future query will
   never involve $d$. Return.

2. Insert $(v, t)$ into $S_0$.

3. If $|S_0| > \alpha$, then discard the element with the earliest
   timestamp in $S_0$, say $t'$. Update $t_0 \leftarrow \max\{t_0, t'\}$

4. Set $i \leftarrow 1$

5. While $(v, t)$ was inserted into level $(i - 1)$ and $i \leq M$,

    (a) Insert $(v, t)$ into $S_i$ with probability 1/2
    (b) If $|S_i| > \alpha$, then discard the element with the
        earliest timestamp in $S_i$, say $t'$. Update $t_i \leftarrow$
        $\max\{t_i, t'\}$
    (c) Increment $i$
```

**Figure 5: Sketch for the Median: When an element $d = (v, t)$ arrives.**

```
1. Let $\ell$ be the smallest integer $0 \leq \ell \leq M$ such that
   $t_\ell < c - w$

2. If $\ell$ does not exist, then return /* Algorithm fails */

3. If $\ell$ exists, then return the median of the set $\{(v, t) \in$
   $S_\ell | t \geq c - w\}$
```

**Figure 6: Sketch for the Median: When an estimate is asked for the median of the elements in timestamp window $[c - w, c]$.**

## 4. UNION OF SKETCHES

In a distributed system, there could be multiple aggregators, each of which is observing a local stream. It may be necessary to compute aggregates on not just any individual stream, but on the union of the data in all the streams. A naive way to solve the problem would be to send all the streams directly to some special *sink* aggregator which would compute a sketch for the union of the streams. However, such an approach would be extremely resource-intensive (in terms of communication cost and energy), since each data item of each stream has to traverse a path from the source to the destination.

A better approach is for each node to compute a sketch of its local stream, and the nodes can communicate their sketches, to estimate the aggregate over the union of all data streams. Since the sketches are much smaller than the streams themselves, this is much more resource-efficient than the naive scheme. In sensor data processing, there have been successful proposals (for example, Madden *et al.* [12]) to combine such sketches in a hierarchical fashion, where the sketches are combined up a spanning tree which is rooted at the sink node (see Figure 7).

Each aggregator sends its sketch to its parent. A parent node receives sketches from its children, combines them into a new sketch and then send the new sketch to its own parent. In this way, the sketches propagate and get combined in

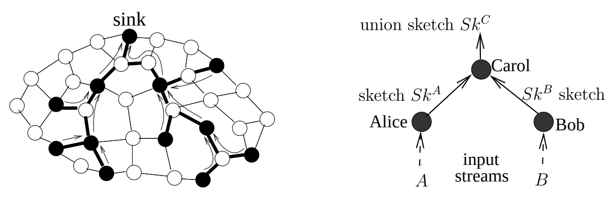

**Figure 7: Left: a spanning tree which connects aggregators with flow of information toward a sink; Right: an aggregator merges the sketches of two aggregators**

the intermediate levels of the tree until they reach the sink. The sink combines the received sketches from its parents and produces a final sketch for the union of all the streams received by all aggregators.

A key property required for the above scheme to work is that it should be possible to combine many sketches into a single one in a *lossless* and *compact* manner. We say that sketches can be combined in a *lossless* manner if the guarantees provided by sketches $A$ and $B$ (for example, $(\epsilon, \delta)$-accuracy for the sum or the median) are preserved when the sketch of the stream $A \cup B$ is computed. We say that sketches can be combined in a *compact* manner if 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. 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.

By using our sketch algorithms for the sum and median, which are based on random samples, we can compute a lossless and compact sketch for the union of all the streams. We consider the simple case of three aggregators, Alice (child), Bob (child), and Carol (parent) (see Figure 7). 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 $A \cup B$.

We first consider the combination of the sketches for the sum. The number of levels in each sketch $Sk^A$ and $Sk^B$ is now $M = \log V_{max}$, where $V_{max}$ is an upper bound on the sum of the observations within the window across all streams (in the single stream case, $V_{max}$ was an upper bound for the sum of all elements in a timestamp window of just one stream). The sketches are combined as follows. Let $S_i^A$ be the level $i$ sample of Alice, where $0 \leq i \leq M$. The sketch for Alice is the vector $Sk^A = \langle S_0^A, S_1^A, \ldots, S_M^A \rangle$. Since $|S_i^A| \leq \alpha$, the size of Alice's sketch is bounded by $|Sk^A| \leq \alpha(M + 1)$. Similarly, $Sk^B = \langle S_0^B, S_1^B, \ldots, S_M^B \rangle$, with $|S_i^B| \leq \alpha$ and $|Sk^B| \leq \alpha(M + 1)$. The sketch for Carol will also consist of $M + 1$ levels of samples $S_i^C$, where $0 \leq i \leq M$, such that $Sk^C = \langle S_0^C, S_1^C, \ldots, S_M^C \rangle$. The level-$i$ samples $S_i^C$ consist of the $\alpha$ most recent (according to their timestamp) elements of the union $S_i^A \cup S_i^B$ of Alice's and Bob's samples at level

$i$. Clearly, $|S_i^C| \leq \alpha$, which implies $|Sk^C| \leq \alpha(M+1)$. Therefore, the combined sketch preserves the same upper bound on its size as the constituent sketches.

We will now show that the combined sketch preserves also the same accuracy as its constituents sketches, but for the union of the streams. For a window size $w \leq W$, (specified during the time of the query), let $R_w^A = \{(v,t) \in A | c-w \leq t \leq c\}$, $R_w^B = \{(v,t) \in B | c-w \leq t \leq c\}$, and $V_A = \sum_{(v,t)\in R_w^A} v$, $V_B = \sum_{(v,t)\in R_w^B} v$, $V = V_A + V_B = \sum_{(v,t)\in(R_w^A)\cup(R_w^B)} v$. That is, $V$ is the sum of the elements in the stream $A \cup B$ that are in the current timestamp window $[c-w, c]$. We have the following.

THEOREM 4.1. *Using $Sk^C$, the returned answer, $X$, for the sum of the elements in the current timestamp window $[c-w, c]$ is an $(\epsilon, \delta)$-estimate of the actual sum $V$, i.e., $\Pr[|X - V| > \epsilon V] < \delta$.*

PROOF. We refer to the proof of Theorem 2.1, and use the same notation that was defined in Section 2.3. If $\ell^\star$ exists, the proof of Theorem 4.1 is exactly same as for the single stream scenario, so we only argue about the existence of $\ell^\star$ in the distributed scenario.

Since $V > \alpha$ (if $V < \alpha$, then an exact answer will be returned by the level 0 sample of $Sk^C$), we have $E[X_0] > \alpha$. At the other extreme, $E[X_M] = V/2^M = V/V_{max} \leq 1$, since we have defined $V_{max}$ as an upper bound on the sum of elements across all streams. For integral $0 \leq i < M$, we have $E[X_i + 1] = E[X_i]/2$. Combining this fact with the values of $E[X_0]$ and $E[X_M]$, we can infer that there must be a level $\ell^\star$ such that $\alpha/4 < E[X_{\ell^\star}] \leq \alpha/2$. $\square$

A similar result to Theorem 4.1 can also be proved for the median. Theorem 4.1 implies that upon combining two sketches, the resulting sketch preserves the desired accuracy for the sum of the union of streams $A$ and $B$. By applying Theorem 4.1 repeatedly for any pair of merged sketches in the spanning tree, it is easy to prove that the final sketch at the sink is an $(\epsilon, \delta)$ estimate for the sum (median) of elements in the union of all the streams. In addition, the size of any produced sketch is bounded by $\alpha(M+1)$.

## 5. REFERENCES

[1] 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.

[2] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 286–296, 2004.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21st ACM Symp. on Principles of Database Systems (PODS)*, pages 1–16, 2002.

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

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

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

[7] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proc. 27th International Conf. on Very Large Data Bases (VLDB)*, pages 541–550, 2001.

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

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

[10] M. Greenwald and S. Khanna. Space efficient online computation of quantile summaries. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 58–66, 2001.

[11] S. Guha, D. Gunopulos, and N. Koudas. Correlating synchronous and asynchronous data streams. In *Proc.9th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 529–534, 2003.

[12] 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.

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

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

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

[16] B. Patt-Shamir. A note on efficient aggregate queries in sensor networks. In *Proc. of the 23rd annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 283–289, 2004.

[17] A. Pavan and S. Tirthapura. Range-efficient computation of $f_0$ over massive data streams. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, 2005.

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

[19] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. 23rd ACM Symposium on Principles of Database Systems (PODS)*, pages 263–274, 2004.