Distrib. Comput. (2008) 20:359–374
DOI 10.1007/s00446-007-0048-7

Sketching asynchronous data streams over sliding windows

Bojian Xu · Srikanta Tirthapura · Costas Busch

Received: 24 August 2006 / Accepted: 4 October 2007 / Published online: 6 November 2007

Abstract We study the problem of maintaining a sketch 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 polylogarithmic with respect to the maximum window size. 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.

Keywords Data streams · Asynchronous streams · Distributed streams · Sliding window · Sum · Median

The work of the authors was supported in part through NSF grants CNS 0520102 and CNS 0520009.
A preliminary version of this paper appeared in Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC) 2006, pages 82–91.
Work done while the third author was at Rensselaer Polytechnic Institute.
Authors are listed in reverse alphabetical order.

B. Xu (✉) · S. Tirthapura
Department of Electrical and Computer Engineering, Iowa State University, 2215, Coover Hall, Ames, IA 50011, USA
e-mail: bojianxu@iastate.edu

S. Tirthapura
e-mail: snt@iastate.edu

C. Busch
Department of Computer Science, Louisiana State University, 280 Coates Hall, Baton Rouge, LA 70816, USA
e-mail: busch@csc.lsu.edu

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" to compute aggregates and statistics, and detect trends and anomalies in traffic. The need for processing such data has led to a 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 [1,3,6,7,9,13] 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 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, 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_is 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 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.

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 where nodes are typically battery operated devices. Unlike previous work [8,9,13] 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 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 a sliding timestamp window over the

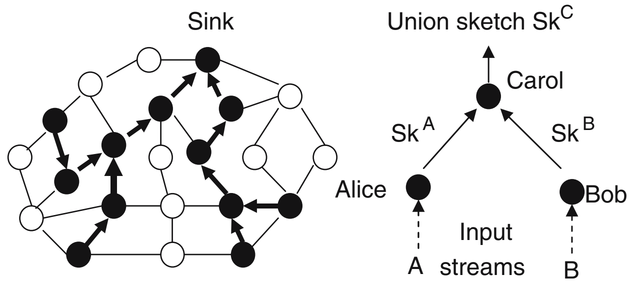

Fig. 1 *Left* a spanning tree which connects aggregators with flow of information towards a sink. *Right* an aggregator merges the sketches of two aggregators

union of all streams (see Fig. 1). 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, $R = (v_1, t_1), (v_2, t_2), \dots, (v_n, t_n)$ is an asynchronous stream of positive integer, timestamp pairs. Let c denote the current time, and W a bound on the maximum window size.

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 $w \le W$, i.e., $V = \sum_{\{(v,t)\in R|c-w\le t\le c\}} v$. 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 (ϵ, δ)-estimator to quantify the quality of answers returned by the algorithm.

Definition 1 For parameters $0 < \epsilon < 1$ and $0 < \delta < 1$, an (ϵ, δ)-estimator for a number Y is a random variable X such that $\Pr[|X - Y| > \epsilon Y] < \delta$. 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 $w \le W$ 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 $O((\frac{1}{\epsilon^2})(\log \frac{1}{\delta})(\log V_{max})\sigma)$, 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 $O(\log \log \frac{1}{\delta} + \log \frac{1}{\epsilon})$.
- Time taken to process a query for the sum is $O(\frac{1}{\epsilon^2} \cdot \log V_{max} \cdot \log \frac{1}{\delta})$

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* [6,9]. Our algorithm solves basic counting immediately by taking $v = 1$ for every data item.

1.1.2 Median

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

Definition 2 For $0 < \epsilon < 1/2$ 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 $(1/2-\epsilon)|S|$ and $(1/2+\epsilon)|S|$ 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 $w \le W$ 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 $O((\frac{1}{\epsilon^2})(\log \frac{1}{\delta})(\log N_{max})\sigma)$, 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 $O(\log \log \frac{1}{\delta} + \log \frac{1}{\epsilon})$.
- Time taken to process a query for the median is $O(\log \log N_{max} + \frac{1}{\epsilon^2} \cdot \log \frac{1}{\delta})$.

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.

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.

1.2 Related work

Datar et al. [6] 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 $O(\frac{1}{\epsilon} \log^2 W)$, 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 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. [6] 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 $\Omega(\log U(\log W + \log U)/\epsilon)$ 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 $O(\log U(\log W + \log U)/\epsilon^2)$ 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 $O(1/\epsilon)$.

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 [6], 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 [5] 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 $O(\log U \log W(\log W + \log U)/\epsilon)$ 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 $1/\epsilon$. Further, their algorithm does not apply to the problem of finding the approximate median.

Arasu and Manku [1] 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. [3] presented algorithms for maintaining the variance and k-medians of elements within a sliding window of a data stream. Feigenbaum et al. [7] considered the problem of maintaining the diameter of a set of points in the sliding window model.

Gibbons and Tirthapura [8] 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 [8], algorithms were presented for estimating the number of distinct elements in the union of distributed streams, and the size of the bit-wise-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 et al. [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 [18] 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 [4], 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 [2,15]. To our knowledge, our work is the first to consider 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_is are positive integers and t_is are the timestamps. Let c denote 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 h ago, in which case setting W to be 24 h will suffice. Note than W can also be set to infinity, which essentially means that the sketch summarizes the whole stream.

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 paper 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 $p = 1, \frac{1}{2}, \frac{1}{4}, \ldots,$. 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 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.

Procedure *SumInit()*

Task: Initialize the sketch.

```
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*/
```

Procedure *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.

```
1. If ($t < c - W$), then Return. /*Discard $d$ since it is outside the largest timestamp window, and a future query will never involve $d$.*/
2. Let $\ell$ = smallest integer $i$, $0 \le i \le M$, such that $v/2^\ell < 1$.
3. Let $r \leftarrow 1$ with probability $v/2^\ell$ and $r \leftarrow 0$ with probability $1 - v/2^\ell$
4. 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 $k \leftarrow 0$.
5. Insert $(v, t)$ into $S_{\ell+k-1}$
6. If $|S_{\ell+k-1}| > \alpha$
   (a) Discard the element with the lowest timestamp in $S_{\ell+k-1}$.
   (b) Let $t'$ be the timestamp of the discarded element.
   (c) $t_{\ell+k-1} \leftarrow \max\{t_{\ell+k-1}, t'\}$
```

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

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

Procedure *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 $[c - w, c]$.

```
1. 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 $\ell'$ exists, then $\ell' \leftarrow M + 1$.
2. If $\ell' \le M$ then
   (a) For $i = \ell'$ to $M$, let $\eta_i \leftarrow \sum_{(v,t) \in S_i, t \ge c-w} \max(\frac{v}{2^{\ell'}}, 1)$
   (b) Return $2^{\ell'} \sum_{i=\ell'}^{M} \eta_i$
3. If $\ell' = M + 1$ then return. /* Algorithm Fails */
```

2.3 Correctness proof

Let X denote the result returned by the procedure *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.

Definition 3 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 $\frac{v}{2^\gamma} < 1$.

– For $0 \le i < \gamma$, $x_i(d) = \frac{v}{2^i}$.
– $x_\gamma(d) = 1$ with probability $\frac{v}{2^\gamma}$, and $x_\gamma(d) = 0$ with probability $1 - \frac{v}{2^\gamma}$.
– 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 $\frac{1}{2}$.

Definition 4 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 5 For $i = 0, \ldots, M$, define $X_i = \sum_{(u,t) \in T_i} u$.

Lemma 1 *If* $d = (v, t)$ *then* $E[x_i(d)] = v/2^i$

Proof Let γ be defined as in Definition 3, 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 3, $\Pr[x_{i+1}(d) = 1] = (1/2) \cdot \Pr[x_i(d) = 1] = v/2^{i+1}$, thus proving the inductive step.

Lemma 2 *For* $i = 0, \ldots, M$, $E[X_i] = \frac{V}{2^i}$

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 1, we get:

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

Lemma 3 *When asked for an estimate for* V, *if the procedure* SumQuery(w) *does not fail in Step 3, then it returns* $2^{\ell'} X_{\ell'}$ *for value* ℓ' *selected in Step 1.*

Proof Consider the procedure *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 the procedure *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 $i \ge \ell'$. 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 $v \ge 2^{\ell'}$. Then, $x_{\ell'}(d) = \frac{v}{2^{\ell'}}$. From step 2(a) in *SumQuery(w)*, the contribution of v to the estimate is $2^{\ell'} \frac{v}{2^{\ell'}} = 2^{\ell'} x_{\ell'}(d)$. 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 2(a) 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 6 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 4 *If $|R_w| \le \alpha$, then the procedure* 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 7 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 5 *Level $\ell^\star$ is uniquely defined and exists for every input stream R.*

Proof From Lemma 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 $V \le 2^M$ 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 a version of Hoeffding bounds from Schmidt et al. [17] (Sect. 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 $Y = y_1 + y_2 + \cdots + y_n$, and let $\mu = E[Y]$.

Lemma 6 Hoeffding's Bound (restated from [17]):

(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 8.

Lemma 7 *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 have $a^{(2^k - 1)} \le a^k < \left(\frac{1}{2}\right)^k$. Therefore, $a^{(2^k)} < \frac{a}{2^k}$.

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

Lemma 8 *For integer ℓ 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 3, 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

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

By the definition of $\ell^\star$, $E[X_{\ell^\star}] > \alpha/4$. Since $E[X_i] = \frac{V}{2^i}$ (from Lemma 2). Using Eq. 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 $\Pr[X_\ell > (1+\epsilon)E[X_\ell]]$

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

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 6 and the fact $(c+\mu_Y)/\mu_Y \ge 1$, we have

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

where we have used $\alpha = 12\frac{\ln 8/\delta}{\epsilon^2}$ and $\delta < 1$, Eq. 2 and Lemma 7.

Case II: $\delta' \ge 1$. Using Lemma 6, we have:

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

where we have used $\alpha = 12\frac{\ln 8/\delta}{\epsilon^2}$ and $\delta < 1$, Eq. 2 and Lemma 7.

From Case I and Case II, we have

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

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

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

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

Using Lemma 6 and the fact $\frac{\mu_Y+c}{\mu_Y} \ge 1$,

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

Using Eq. 2 and Lemma 7,

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

Thus, we have

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

Combining Eqs. 3 and 4, for $0 \le \ell \le \ell^\star$, we get

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

Lemma 9

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

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

Using Lemma 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 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 the procedure *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 $i = 0\ldots M$, $|T_i| \ge \beta_i$, 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 4, it follows that for all $0 \le i_1 < i_2 \le M$, $|T_{i_1}| \ge |T_{i_2}|$. Thus, if $\ell'-1 \ge \ell^\star$, then $|T_{\ell^\star}| \ge |T_{\ell'-1}| > \alpha$.

$$\Pr[\ell' > \ell^\star] = \Pr[\ell'-1 \ge \ell^\star] \le \Pr[|T_{\ell^\star}| > \alpha] \quad (5)$$

Since each element in T_i contributes at least one to X_i, we have $X_i \ge |T_i|$. Combining this with Eq. 5, we get:

$$\Pr[\ell' > \ell^\star] \le \Pr[X_{\ell^\star} > \alpha] \quad (6)$$

As in the proof of Lemma 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 $E[X_{\ell^*}] \le \alpha/2$, we have

$$\Pr[X_{\ell^\star} > \alpha] \le \Pr[X_{\ell^\star} > 2E[X_{\ell^\star}]] = \Pr[c+Y > 2(c+\mu_Y)] = \Pr\left[Y > \mu_Y\left(1+\frac{c+\mu_Y}{\mu_Y}\right)\right]$$

Using Lemma 6,

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

where $\delta' = \frac{c+\mu_Y}{\mu_Y} > 1$.

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

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

Theorem 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.

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

where we have used Lemmas 9 and 10.

2.4 Complexity

Lemma 11 Space complexity: *The total space taken by the sketch for the sum is $O((\frac{1}{\epsilon^2})(\log\frac{1}{\delta})(\log V_{max})\sigma)$, 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 $\alpha = 12\frac{\ln(8/\delta)}{\epsilon^2}$ 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 12 Time complexity: *The worst case time complexity for processing an element $d = (v, t)$ by* SumProcess(d) *is $O(\log \alpha) = O(\log \log \frac{1}{\delta} + \log \frac{1}{\epsilon})$. The worst case time taken to answer a query for the sum by* SumQuery(w) *is $O(M\alpha) = O(\frac{1}{\epsilon^2} \cdot \log V_{max} \cdot \log \frac{1}{\delta})$.*

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

Consider the procedure *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 5, and (possibly) discarding the oldest element of $S_{\ell+k-1}$ in Step 6, 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, $\ell \le i \le M$. This part takes time $O(M\alpha)$. Summing these two parts, the worst case time taken for answering a query is $O(M\alpha)$.

2.5 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 5, it can be inserted into every level starting from 0 till $(\ell + k - 1)$ (Fig. 2 of [19]). This way, when processing a query for the sum in *SumQuery*, we need to consider only a single level ℓ' (Fig. 3 of [19]), 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 $O((\log V_{max})(\log \log 1/\delta + \log 1/\epsilon))$, and time taken to answer a query for the sum is $O(\log \log V_{max} + \frac{\log 1/\delta}{\epsilon^2})$. 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, $0 \le L < M$, as a parameter to procedures *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 $L, L+1, \ldots, \ell + k - 1$. Procedure *SumQuery* is modified as follows. As before, level $\ell' \in [0, M]$ is the smallest integer such that for all j, $\ell' \le j \le M$, $t_j < c - w$. 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 $\ell' \ge L$, 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 $O\left((\lceil \log V_{max} \rceil - L)(\log \log \frac{1}{\delta} + \log \frac{1}{\epsilon})\right)$, and worst case time for answering a query for the sum is $max\ (O\ (\log(\lceil \log V_{max} \rceil - L) + \frac{\log 1/\delta}{\epsilon^2}),\ O\left(L \cdot \frac{\log 1/\delta}{\epsilon^2}\right))$. 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 Fig. 2 of [19], 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 Fig. 3 of [19]; if we choose $L = M - 1$, it is procedure *SumProcess* in Sect. 2.2 which process an element, and the algorithm for answering a query for the sum is the procedure *SumQuery* in Sect. 2.2.

3 Computing the median

In this section, given a maximum window size W, we design a sketch such that for all $w \le W$, 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 [10, 14]. 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.

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 $\alpha = \frac{96}{\epsilon^2} \ln(\frac{8}{\delta})$. Initially, each S_i is empty and t_i is set to be -1, as described in the procedure *SumInit*. The algorithm for updating the sketch upon receiving a new element is described in procedure *MedianProcess*, and procedure *MedianQuery* returns an estimate of the median of R_w when receiving a query.

3.2 Correctness proof

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

Procedure *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 $(t < c - W)$, Return.
2. Insert $(v, t)$ into $S_0$.
3. If $|S_0| > \alpha$
   (a) Discard the element with the earliest timestamp in $S_0$, say $t'$.
   (b) Update $t_0 \leftarrow \max\{t_0, t'\}$
4. Set $i \leftarrow 1$
5. While $(v, t)$ was inserted into level $(i-1)$ and $i \le M$,
   (a) Insert $(v, t)$ into $S_i$ with probability $1/2$
   (b) If $|S_i| > \alpha$
       i. Discard the element with the earliest timestamp in $S_i$, say $t'$.
       ii. Update $t_i \leftarrow \max\{t_i, t'\}$
   (c) Increment $i$
```

Procedure *MedianQuery(w)*

Input: $w \le W$ 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 $\ell'$ be the smallest integer $0 \le \ell' \le M$ such that $t_{\ell'} < c - w$
2. If $\ell'$ exists, then return the median of the set $\{(v, t) \in S_{\ell'} | t \ge c - w\}$
   Else $\ell' \leftarrow M + 1$
3. If $\ell' = M + 1$, then return /* Algorithm fails */
```

Definition 8 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 $\frac{1}{2}$. If $x_{i-1}(d) = 0$, then $x_i(d) = 0$.

Definition 9 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 13 *Given any $d = (v, t)$, for each i, $0 \le i \le M$, $E[x_i(d)] = 1/2^i$, $E[X_i] = \frac{N}{2^i}$.*

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 8. Assume for $0 \le i < M$, $\Pr[x_i(d) = 1] = 1/2^i$. Using Definition 8, $\Pr[x_{i+1}(d) = 1] = \frac{1}{2} \cdot \Pr[x_i(d) = 1] = \frac{1}{2^{i+1}}$, proving the inductive step.

Now we show $E[X_i] = \frac{N}{2^i}$. Note that $|R_w| = N$. The Definitions 8 and 9 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 14 *When asked for an estimate for the median, if* MedianQuery(w) *does not fail in Step 3, 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 procedure *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 procedure *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 9, 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 10 For $i = 0 \dots M$, let r_i denote the rank of γ_i in R_w. Define event B_i to be true if if $r_i \notin [(1/2 - \epsilon)N, (1/2 + \epsilon)N]$, and false otherwise. Define event G_i to be true if $(1-\epsilon)\frac{N}{2^i} \leq X_i \leq (1+\epsilon)\frac{N}{2^i}$, and false otherwise. Let $\ell^\star \geq 0$ be an integer such that $\alpha/4 < E[X_{\ell^\star}] \leq \alpha/2$.

Lemma 15 *Level $\ell^\star$ is uniquely defined and exists for every input stream R.*

Proof From Lemma 13, 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 16 *For $0 \leq \ell \leq \ell^*$,*

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

Proof

$$\begin{aligned}\Pr[B_\ell] &= \Pr[G_\ell \wedge B_\ell] + \Pr[\bar{G}_\ell \wedge B_\ell] \\ &\leq \Pr[B_\ell | G_\ell] \cdot \Pr[G_\ell] + \Pr[\bar{G}_\ell] \qquad (7) \\ &\leq \Pr[B_\ell | G_\ell] + \Pr[\bar{G}_\ell] \qquad (8)\end{aligned}$$

Using Lemmas 17 and 18 in Eq. 8, we get:

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

Lemma 17 *For $0 \leq \ell \leq \ell^*$,*

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

Proof Let $\mu_\ell = E[X_\ell] = \frac{N}{2^\ell}$

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

Since $E[X_i] = \frac{N}{2^i}$ (from Lemma 13) and $E[X_{\ell^*}] > \alpha/4$, we have $\mu_\ell > \frac{\alpha}{4} 2^{\ell^*-\ell}$. By Definition 2, we know $0 < \epsilon < \frac{1}{2}$. Using Lemma 6,

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

We have used Lemma 7 in the last inequality.

Lemma 18 *For $0 \leq \ell \leq \ell^*$,*

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

Proof

$$\Pr[B_\ell | G_\ell] = \Pr\left[r_\ell < \left(\frac{1}{2} - \epsilon\right) N | G_\ell\right] + Pr\left[r_\ell > \left(\frac{1}{2} + \epsilon\right) N | G_\ell\right]$$

The proof will consist of two parts, Eqs. 9 and 10.

$$\Pr\left[r_\ell < \left(\frac{1}{2} - \epsilon\right) N | G_\ell\right] < \frac{\delta/4}{2^{\ell^*-\ell+2}} \qquad (9)$$

$$\Pr\left[r_\ell > \left(\frac{1}{2} + \epsilon\right) N | G_\ell\right] < \frac{\delta/4}{2^{\ell^*-\ell+2}} \qquad (10)$$

Proof of Eq. 9: Let $L = \{d \in R_w | \text{rank of } d \text{ in } R_w \leq (\frac{1}{2} - \epsilon)N\}$, $Y = \sum_{d \in L} x_\ell(d)$. By Lemma 13, we have $E[Y] = (1 - 2\epsilon)\frac{N}{2^{\ell+1}}$ Since $r_\ell < (\frac{1}{2} - \epsilon)N$, which means that at least the smaller half elements in T_i were selected from the set L, combining the fact $X_i \geq (1-\epsilon)\frac{N}{2^\ell}$, we have the following,

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

where $\delta' = \frac{\epsilon}{1-2\epsilon}$ and $\Pr[G_\ell] \geq 1 - \frac{\delta}{8\cdot 2^{\ell^*-\ell+2}}$

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

$$\begin{aligned}
\Pr[Y \geq (1+\delta')E[Y]] &\leq e^{-E[Y]\delta'^2/3} < \left(\frac{\delta}{8}\right)^{\frac{2^{\ell^*-\ell+2}}{1-2\epsilon}}\\
&< \left(\frac{\delta}{8}\right)^{2^{\ell^*-\ell+2}} \leq \frac{\delta/8}{2^{\ell^*-\ell+2}}
\end{aligned}$$

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

$$\begin{aligned}
\Pr[Y \geq (1+\delta')E[Y]] &\leq e^{-E[Y]\delta'/3} < \left(\frac{\delta}{8}\right)^{\frac{2^{\ell^*-\ell+2}}{\epsilon}}\\
&< \left(\frac{\delta}{8}\right)^{2^{\ell^*-\ell+2}} \leq \frac{\delta/8}{2^{\ell^*-\ell+2}}
\end{aligned}$$

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 $E[X_\ell] = \frac{N}{2^\ell} > \frac{\alpha}{4}2^{\ell^*-\ell}$ in addition to Lemma 6 and Lemma 7. From Cases 1 and 2, Eq. 9 is proved. Equation 10 can be similarly proved. From Eqs. 9 and 10, we get:

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

Lemma 19

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

Proof The proof directly follows from Lemma 16

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

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

Lemma 20

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

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 6, we have

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

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

Theorem 2 *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 19 and 20 and a similar argument to the Proof of Theorem 1, we get:

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

3.3 Complexity

Lemma 21 Space complexity: *The total space taken by the sketch for the median is* $O((\frac{1}{\epsilon^2})(\log\frac{1}{\delta})(\log N_{max})\sigma)$*, 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 $\alpha = 96\frac{\ln(8/\delta)}{\epsilon^2}$ 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 22 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* $O\big(\log\log N_{max} + \frac{\log(1/\delta)}{\epsilon^2}\big)$

Proof The proof is similar to that of Lemma 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 $O(\log \log N_{max} + \alpha)$.

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.

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. [12]) to combine such sketches in a hierarchical fashion, where sketches are combined up a spanning tree which is rooted at the sink node (see Fig. 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 Fig. 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 $A \cup B$.

```
Procedure Union($Sk^A$, $Sk^B$)

Input:
1. $Sk^A = \langle S_0^A, t_0^A, S_1^A, t_1^A, \ldots, S_M^A, t_M^A \rangle$, a sketch of Alice's local stream $A$
2. $Sk^B = \langle S_0^B, t_0^B, S_1^B, t_1^B, \ldots, S_M^B, t_M^A \rangle$, a sketch of Bob's local stream $B$
Output: $Sk^C$, a sketch of $C = A \cup B$

For every $i = 0 \ldots M$
1. If $|S_i^A \cup S_i^B| \leq \alpha$, then
    (a) $S_i^C \leftarrow S_i^A \cup S_i^B$
    (b) $t_i^C \leftarrow \max\{t_i^A, t_i^B\}$
2. Else,
    (a) $S_i^C$ is the set of $\alpha$ most recent elements in $S_i^A \cup S_i^B$
    (b) Let $t$ be the most recent timestamp in
        $\big((S_i^A \cup S_i^B) - S_i^C\big)$.
        Then $t_i \leftarrow \max\{t, t_i^A, t_i^B\}$.
```

The algorithm for the union is formally described in Procedure *Union*$(\cdot, \cdot)$. Given sketches Sk^A and Sk^B of streams A and B, respectively, the procedure 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 Procedure *Union()*, for $0 \leq i \leq M$, let S_i^A denote the level i sample of Alice, and t_i^A denote 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 $\langle S_0^C, t_0^C, S_1^C, t_1^C, \ldots, S_M^C, t_M^C \rangle$. 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 11. Similarly, the sketch for the median is compact, and has a space complexity as described in Lemma 21.

Losslessness of sketches. We will now show that the sketch resulting from the *Union()* procedure 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 Sect. 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 Sect. 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* (Sect. 2) and *MedianQuery* (Sect. 3) will never consider such elements with timestamps less than $(c - W)$, even if they are present in the sketch.

Lemma 23

$$Sk_{sum}^C = Sk_{sum}^{A \cup B}$$
$$Sk_{med}^C = Sk_{med}^{A \cup B}$$

Proof We show $Sk_{sum}^C = Sk_{sum}^{A \cup B}$, and a similar argument holds for $Sk_{med}^C = Sk_{med}^{A \cup B}$. Let R_W^A and R_W^B 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 $S_{sum,i}^A$ denote the ith level sample of Sk_{sum}^A. Similarly, we define $S_{sum,i}^B$, $S_{sum,i}^C$ and $S_{sum,i}^{A \cup B}$.

For any element $(v, t) \in R_W^A \cup R_W^B$, note that the same procedure *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 $(v, t) \in R_W^A$ is selected into $S_{sum,i}^A$, then it will also be selected into $S_{sum,i}^{A \cup B}$, and vice versa. Therefore the set of elements that are ever selected into $S_{sum,i}^A$ or $S_{sum,i}^B$ is exactly the same set of elements that are ever selected into $S_{sum,i}^{A \cup B}$. For any level $i = 0 \ldots M$, $S_{sum,i}^A$ retains the α elements with the most recent timestamps that were ever selected into $S_{sum,i}^A$, and similarly with $S_{sum,i}^B$. From Steps (1) and (2) of procedure *Union()*, we see that $S_{sum,i}^C$ retains the α most recent elements that ever selected into $S_{sum,i}^A$ or into $S_{sum,i}^B$. Thus, $S_{sum,i}^C$ retains the α most recent elements among $A \cup B$ that were selected into level i of Sk^A or Sk^B.

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

From the above lemma, it follows that all properties of $Sk_{sum}^{A \cup B}$ carry over to Sk_{sum}^C. From Theorem 1 we know $Sk_{sum}^{A \cup B}$ provides an (ϵ, δ)-estimate for the sum within any timestamp window of width at most W on $A \cup B$. Thus, Sk_{sum}^C 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 23 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.

5 Conclusions

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.

We note that there is an $O(1/\epsilon)$ factor gap between our upper bound on the space complexity of maintaining the approximate sum, and the known lower bound from [6]. An open problem is to close this gap through either improved algorithms or better lower bounds.

References

1. Arasu, A., Manku, G.: Approximate counts and quantiles over sliding windows. In: Proceedings of ACM Symposium on Principles of Database Systems (PODS), pp. 286–296 (2004)
2. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS), pp. 1–16 (2002)
3. Babcock, B., Datar, M., Motwani, R., O'Callaghan, L.: Maintaining variance and k-medians over data stream windows. In: Proceedings of 22nd ACM Symposium on Principles of Database Systems (PODS), pp. 234–243 (2003)
4. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
5. Busch, C., Tirthapura, S.: A deterministic algorithm for summarizing asynchronous streams over a sliding window. In: Proceedings of International Symposium on Theoretical Aspects of Computer Science (STACS), pp. 465–476 (2007)
6. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows. SIAM J. Comput. **31**(6), 1794–1813 (2002)
7. Feigenbaum, J., Kannan, S., Zhang, J.: Computing diameter in the streaming and sliding-window models. Algorithmica **41**, 25–41 (2005)
8. Gibbons, P., Tirthapura, S.: Estimating simple functions on the union of data streams. In: Proceedings of ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 281–291 (2001)
9. Gibbons, P., Tirthapura, S.: Distributed streams algorithms for sliding windows. Theory Comput. Syst. **37**, 457–478 (2004)
10. Greenwald, M., Khanna, S.: Space efficient online computation of quantile summaries. In: Proceedings of ACM International Conference on Management of Data (SIGMOD), pp. 58–66 (2001)
11. Guha, S., Gunopulos, D., Koudas, N.: Correlating synchronous and asynchronous data streams. In: Proceedings of 9th ACM International Conference on Knowledge Discovery and Data Mining (KDD), pp. 529–534 (2003)
12. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: Tag: a tiny aggregation service for ad-hoc sensor networks. SIGOPS Oper. Syst. Rev. **36**(SI), 131–146 (2002)
13. Manjhi, A., Shkapenyuk, V., Dhamdhere, K., Olston, C.: Finding (recently) frequent items in distributed data streams. In: Proceedings of IEEE International Conference on Data Engineering (ICDE), pp. 767–778 (2005)
14. Manku, G., Rajagopalan, S., Lindsley, B.: Approximate medians and other quantiles in one pass and with limited memory. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 426–435 (1998)
15. Muthukrishnan, S.: Data streams: algorithms and applications. technical report. Rutgers University, Piscataway (2003)
16. Patt-Shamir, B.: A note on efficient aggregate queries in sensor networks. In: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 283–289 (2004)
17. Schmidt, J., Siegel, A., Srinivasan, A.: Chernoff-hoeffding bounds for applications with limited independence. SIAM J. Discrete Math. **8**(2), 223–250 (1995)
18. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: Proceedings of 23rd ACM Symposium on Principles of Database Systems (PODS), pp. 263–274 (2004)
19. Tirthapura, S., Xu, B., Busch, C.: Sketching asynchronous streams over a sliding window. In: Proceedings of the 25th Annual ACM Symposium on Principles of Distributed domputing(PODC), pp. 82–91 (2006)