Algorithmic Techniques for Processing Data Streams *

Elena Ikonomovska[1] and Mariano Zelke[2]

1 **Jožef Stefan Institute**
Jamova cesta 39, 1000 Ljubljana, Slovenia
elena.ikonomovska@ijs.si

2 **Institute for Computer Science**
Goethe-University
60325 Frankfurt am Main, Germany
zelke@em.uni-frankfurt.de

Abstract

We give a survey at some algorithmic techniques for processing data streams. After covering the basic methods of sampling and sketching, we present more evolved procedures that resort on those basic ones. In particular, we examine algorithmic schemes for similarity mining, the concept of group testing, and techniques for clustering and summarizing data streams.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases streaming algorithm, sampling, sketching, group testing, histogram

Digital Object Identifier 10.4230/DFU.Vol5.10452.237

1 Introduction

The opportunity to automatically gather information by myriads of measuring elements proves to be both a blessing and a challenge to science. The volume of available data allows a problem examination to be more profound than ever before; climate prediction [31] and particle physics [17] are unthinkable without exploring mounds of data. However, the challenge is posed by the necessity to inspect these amounts of information. The particle physics experiment at the large hadron collider of CERN will soon produce data of a size of 15 petabytes annually [51] corresponding to more than 28 gigabytes on average every minute.

This challenge puts the basic principle of the traditional RAM-model, cf. [3], in question: It is unreasonable to take a main memory for granted that includes the whole input and allows fast random access to every single input item. On the contrary, for applications as the above ones massive input data must be processed that goes beyond the bounds of common main memories and can only be stored completely on external memory devices. Since random access is very time-consuming on these devices, traditional algorithms depending on random access show unfeasible running times.

Streaming algorithms drop the demand of random access to the input. Rather, the input is assumed to arrive in arbitrary order as an *input stream*. Moreover, streaming algorithms are designed to settle for a working memory that is much smaller than the size of the input.

Because of these features, streaming algorithms are the method of choice if emerging data must be processed in a real-time manner without completely storing it. In addition,

* This work was partially supported by the project "Foundations of data stream processing at FAIR computing" and the Slovenian Research Agency.

streaming algorithms can also benefit from their properties when processing data that is stored on large external memory devices. Compared to their slow random access, the output rates of such devices grow by magnitudes when the data content is dispensed in the order it is stored, i. e., as a stream that can be handled by a streaming algorithm.

There is a large variety of streaming algorithms. They vary in several aspects such as the number of passes that are permitted over the input stream, the size of the consumed memory, or the time required to process a single input item. Such algorithms can be deterministic or randomized, they might process streams comprising of numerical values, a graph's edges, coordinates of points, or parts of an XML document. For an overview of the rich literature on streaming algorithms, we refer the reader to [9] and [58].

This work covers basic algorithmic techniques that are utilized by a multitude of streaming algorithms. These techniques often form the foundation to which more sophisticated methods revert to. After giving the necessary definitions in Section 2, we present the techniques of sampling and sketching in Section 3 and Section 4, respectively. Then we build upon these ideas and present some more advanced algorithmic techniques for similarity mining in Section 5, the concept of group testing and its application to tracking hot items in Section 6, and techniques for clustering and summarizing data streams based on robust approximation in Section 7.

2 Preliminaries

Let U be a universe of n elements. Even though the members of U can be any objects, it is convenient to identify them with natural numbers, thus, we assume $U = \{1, 2, \ldots, n\}$. There are several kinds of *streams*; the most natural one is simply a sequence $a_1, a_2, \ldots, a_\omega$ of ω items where each item is an element of U. Elements of U may occur once in the stream, several times, or not at all. The sequence $2, 1, 2, 5$ serves as an example. Such a sequence is called a stream in the *cash register model*. Streams of the cash register model are widely considered in practice; a sequence of IP addresses that access a web server is a typical instance.

For an element j in our universe U, we can consider the number of occurrences of j in the cash register stream. This way, we get the *frequency* of j that is denoted as f_j. More formally, $f_j = |\{i : a_i = j,\ 1 \leq i \leq \omega\}|$, that is, the number of positions in the stream at which element j appears. If we know the frequency for every element in U, we can arrange a *frequency vector* $f = (f_1, f_2, \ldots, f_n)$ containing the number of occurrences in the stream for each $j \in U$ at the corresponding position. For our example stream, the frequency vector is $(1, 2, 0, 0, 1, 0, \ldots, 0)$ containing a zero for every element of U that is not part of the stream.

If we read a cash register stream item-wise from left to right, we can perceive this as gradual updates to the frequency vector of U: Starting with the all-zero n-dimensional vector, every a_i in the stream causes an increment of the corresponding vector entry by one. After processing the whole stream this way, the frequency vector emerges.

It is not hard to figure a generalization of the described update scheme: Instead of a single element $j \in U$ as a stream item incrementing f_j by one, we can imagine a stream item $(j, z) \in U \times \mathbb{Z}$. Such a pair in the stream changes f_j by the amount of z, i. e., f_j is increased or decreased by $|z|$ depending on the sign of z. A stream composed of such pairs is called *turnstile model* stream.

A turnstile stream represents a frequency vector f of U since it describes f_j for each $j \in U$ as the sum of all changes in the stream that are made on j. If for every prefix of the stream the represented vector consists of nonnegative entries only, we call this the *strict turnstile model.* The sequence $(3, 4), (2, 2), (5, 2), (1, 1), (5, -1), (3, -4)$ is an example for a strict turnstile stream that gives rise to the same frequency vector of U as the previously mentioned cash register example stream.

For the *non-strict turnstile model*, we allow the represented frequency vector to have negative entries as well. An example is given by the sequence $(2, -1)$,$(5, 1)$,$(3, -3)$,$(2, 3)$,$(1, 1)$,$(3, 3)$ representing the same frequency vector as previous examples.

It easy to imagine a strict turnstile stream as a sequence of insert/delete operations to a database. Every item (j, z) with positive (negative) z corresponds to inserting (deleting) item j $|z|$ times into (from) the database. The strict case applies here because at every moment no entry is deleted from the database that has not been inserted before. We will see the usage of this model in Section 6 when tracking frequent items in a database. As it turns out in Section 5, the non-strict model has applications when examining the similarity of two streams.

For some applications, it is common to use a certain restriction of the turnstile model. In the turnstile model, the stream is a sequence $(a_1, z_1), (a_2, z_2), \dots, (a_\omega, z_\omega)$ of pairs. Now let us assume that for each element in U there is exactly one pair in the stream and the pairs are ordered by the first component, that is, $a_i = i$ for every pair. Thus, we get a stream like $(1, z_1), (2, z_2), \dots, (n, z_n)$ of n pairs. Since every a_i is defined by its position in the stream, we can drop the a_i's and end up with a stream $z_1, z_2, \dots, z_n$. Such a stream is called a *time series model* stream and it represents a frequency vector of U in the most elementary way: It is just a sequence of the frequency vector entries written from left to right, i. e., $f_j = z_j$ for every $j \in U$.

The only time series stream possible representing the same frequency vector as previous example streams is the sequence $1, 2, 0, 0, 1, 0, \dots, 0$ of length n. The time series model has applications in areas like sensor networks or stock markets where periodical updates like measurements or share values are monitored. Each input item gives the observed value at the corresponding moment and characteristics of the stream describe the value's behavior.

For any given stream $a_1, a_2, \dots, a_\omega$, a *streaming algorithm* reads the stream item by item from left to right. It is forbidden to have random access to the stream. For the cash register and turnstile model, such an algorithm cannot make any assumptions on the item's order, that is, it must be prepared for any order. Furthermore, the size of the memory for a streaming algorithm is restricted: It must be sublinear in the size n of the universe and sublinear in the *cardinality* of the stream which is defined as $\sum_{j \in U} |f_j|$. We denote this cardinality by m. Notice that for the cash register model, m equals ω, i. e., the number of items in the stream. Hence, we will often write $a_1, a_2, \dots, a_m$ for an input stream in the cash register model omitting the ω. For the strict turnstile model, m is the total number of items that have been inserted and not deleted.

Apparently, we assumed all our streams to be finite, that is, we have a first item a_1 and a last item a_ω or a_m. That seems to contradict many applications; sequences of IP addresses accessing a web server or streams of operations to a database do not have a predefined end. However, from the perspective of a streaming algorithm—and this very perspective we take—the end of the stream is reached when the algorithm is queried about the stream. At this moment, the last item is fixed, that is, the finite initial segment of a potentially infinite stream is determined and framed as the object of investigation. However, the precise end of the stream may not be known in advance which serves as a challenge for a streaming algorithm that must must be prepared for answering a query about the stream at any moment.

It is important to note that this broad definition of a streaming algorithm spans a large spectrum of algorithms. There are streaming algorithms consuming a memory that is polynomially smaller than the input size, e. g. [42], others are content with a polylogarithmic amount, e. g. [6]. While one-pass algorithms [60] are designed for a single run over the input stream, there are also algorithms that read the input stream several times. Some of those multi-pass algorithms assume the input stream to be unchanged between different passes, e. g. [19], others have the ability to influence the order of the input items prior to every pass, e. g. [2]. However, in this work we restrict ourselves to the case of one-pass algorithms.

Since most streaming algorithms work in a randomized fashion, we utilize tools from probability theory for their presentation. For an introduction to this area as well as for definitions and properties of terms as expectation and variance, further for inequalities due to Markov, Chebyshev, and Chernoff, we refer the reader to [55].

3 Sampling

Generally, *sampling* denotes a rule-based process that selects a smaller number of items out of a larger group. It is easy to see that such an approach can be useful in the streaming context. In particular, if we assume the cash register model—and that is what we do for the whole section—the sampling approach smoothly applies: Out of the large group of all items $a_1, a_2, \dots, a_m$ in the input stream, the algorithm chooses a group of size smaller, in most cases much smaller, than m to be kept in memory to consume a space sublinear in m. The idea is that at the end of the stream or whenever the algorithm is queried, it uses the memorized items, that is, the sample, to gain information about the whole stream. Of course, the accuracy of this information heavily depends on how well the sample represents the whole stream according to the query. We will see instances of representative samples in Section 3.1. To draw a characteristic sample is the challenge for any sampling approach.

Though there are some deterministic sampling methods in the area of streaming algorithms, e. g. [36, 62], the predominant part of sampling approaches in this area is randomized and hence subject of this chapter.

3.1 Reservoir Sampling

Assume we want to sample from the input stream $a_1, a_2, \dots, a_m$ a single item s uniformly at random, that is, in such a way that the probability of being the sample is the same for every input item. Hence, we require $Pr[a_\ell \text{ is the sample } s] = 1/m$ for $1 \leq \ell \leq m$.

It is important to note here that we draw a uniform sample over all *input items* and not over the *elements* of the universe U. Hence, for our sampling purposes, we make a difference between two input items a_i and a_j as long as $i \neq j$, even if a_i and a_j denote the same element of U.

From the input stream $2, 1, 2, 5$ for example, we want to pick one of the four input items uniformly at random, that is, each with probability $1/4$, and we do not care that two of those items represent the same element of U. Of course, the universe element 2 is chosen as the sample with probability $2/4$ because of the two corresponding input items a_1 and a_3 while the universe elements 1 and 5 are each sampled with probability $1/4$. Indeed, this is intended since the element 2 occurs as twice as often as each of 1 and 5.

We see that an element's frequency of occurrence proportionally affects the probability for being the sample. Therefore, by drawing and examining samples we can try to deduce information about the frequency distribution of the input stream; an example for doing so is given later in this subsection.

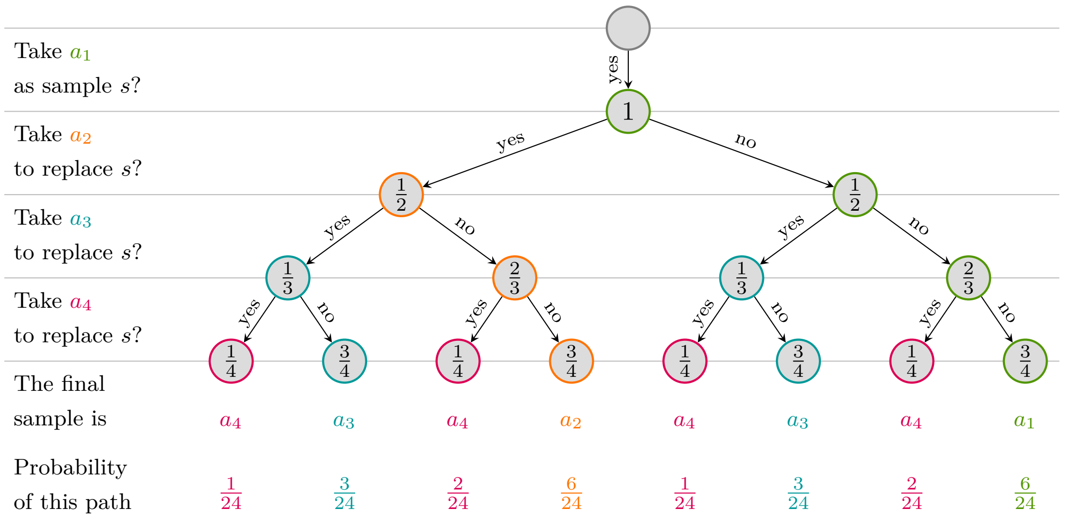
Take a_1 as sample s?
Take a_2 to replace s?
Take a_3 to replace s?
Take a_4 to replace s?
The final sample is
Probability of this path
yes
no
1
$\frac{1}{2}$
$\frac{1}{3}$
$\frac{2}{3}$
$\frac{1}{4}$
$\frac{3}{4}$
a_4 a_3 a_4 a_2 a_4 a_3 a_4 a_1
$\frac{1}{24}$ $\frac{3}{24}$ $\frac{2}{24}$ $\frac{6}{24}$ $\frac{1}{24}$ $\frac{3}{24}$ $\frac{2}{24}$ $\frac{6}{24}$

Figure 1 Decision tree for the reservoir sampling algorithm of stream a_1, a_2, a_3, a_4. The algorithm randomly decides to take or not to take ("yes" or "no") the considered input item as the actual sample s. Each node is labeled by the probability of the previous decision and its color indicates the item currently chosen as s. The probability of a specific path through this tree results from multiplying the probabilities along this path.

If we want to sample an input item uniformly at random from the stream and the length m of the stream is known in advance, this is a very simple task: Before reading the stream, the algorithm chooses a number $\ell \in \{1, 2, \ldots, m\}$ uniformly at random; then it reads the stream until item a_ℓ which is picked as the sample. For the space consumption we note that the algorithm needs to generate ℓ and to memorize ℓ and a_ℓ, additionally it requires to count the number of stream items up to ℓ. Since a memory of $\mathcal{O}(\log m + \log n)$ suffices to do so and the stream is accessed sequentially, this method in fact describes a streaming algorithm using one pass.

However, the prior knowledge of m is a fairly unrealistic assumption. On the contrary, in most streaming scenarios the length of the stream is unknown beforehand or—even worse—there is no pre-defined end of the stream. Such continuous streams can arise from perpetual sensor updates; here, the unpredictable moment of a query marks the end of a stream on which the query needs to be evaluated.

It might come as a surprise that we are able to draw a uniform random sample without knowing the length of the stream. This approach is called *reservoir sampling* and is due to Vitter [63]. For every position ℓ in the stream $a_1, a_2, \ldots, a_m$, it maintains an item s that is a uniform random sample over all items a_i, $i \leq \ell$, that is, over all items of the stream up to a_i. At the end of the stream, s is the final sample drawn from the whole input stream.

The algorithm starts by setting a_1 as s. In the following step, a_2 is chosen to replace a_1 as the sample with probability $1/2$; next, a_3 is picked as s with probability $1/3$. In general, for $i \geq 2$, a_i is memorized as s—and thereby replaces the previously stored item—with probability $1/i$.

Figure 1 shows the decision tree of the reservoir sampling algorithm for a stream of four input items. Each non-leaf node corresponds to a random decision of the algorithm whether or not to replace the actual sample s by the current input item. Every node is labeled by the probability of the preceding decision. Hence, the product of all labels along a path from the root to a leaf gives the probability for the specific sequence of decisions corresponding to this path.

As an example we calculate the probability for choosing the second input item a_2 as the final sample from the stream a_1, a_2, a_3, a_4. After reading the first input item, the algorithm chose a_1 as the actual sample. While reading a_2 in the next step, the algorithm picks a_2 as s with a probability of 1/2. To end up with a_2 as the final sample, the algorithm has to decide to select neither a_3 nor a_4 as the actual sample in the next two steps. The item a_3 is not picked with a probability of $(1 - \frac{1}{3}) = 2/3$; a_4 is not chosen with a probability of $(1 - \frac{1}{4}) = 3/4$. Item a_2 ends up as the final sample if and only if all mentioned events occur which happens with a probability of $\frac{1}{2} \cdot \frac{2}{3} \cdot \frac{3}{4} = \frac{1}{4}$. Similarly, one can calculate the same probability for the selection of each other input item as the final sample.

Another point of view on our small example is given by Figure 1: The probability just calculated for choosing a_2 as the final sample s corresponds to the decision sequence "yes", "yes", "no", "no" and thus to the path from the root to the orange leaf. However, while a_1 and a_2 each have a single corresponding leaf only, a_3 and a_4 correlate with several leafs, that is, decision sequences. Of course, the probability to end up with such items as the final s is the sum over the probabilities of the corresponding sequences. Eventually, we get a probability of 1/4 for every of the four input items.

The described reservoir sampling algorithm is certainly a streaming algorithm as it sequentially reads the input stream and only requires to memorize a single item. To convince ourselves that the choice for s at the end of the stream yields a uniform random sample, we look at the probability that some a_ℓ, $1 \le \ell \le m$, is the final s. This happens if a_ℓ is chosen as the actual s and additionally none of $a_{\ell+1}, a_{\ell+2}, \dots, a_m$ replaces a_ℓ as the actual sample. For this probability, we have

$$\begin{aligned}
Pr[\, a_\ell \text{ is the final } s\,] &= Pr[\, a_\ell \text{ is chosen as } s\,] \cdot \prod_{i=\ell+1}^{m} Pr[\, a_i \text{ does not replace } a_\ell \text{ as } s\,] \\
&= \frac{1}{\ell} \cdot \prod_{i=\ell+1}^{m} \left(1 - \frac{1}{i}\right) \\
&= \frac{1}{\ell} \cdot \prod_{i=\ell+1}^{m} \frac{i-1}{i} \\
&= \frac{1}{m}
\end{aligned}$$

which means that any item a_ℓ ends up as the final sample with the same probability.

It is not hard to imagine a situation where we want to have a sample from the stream that is larger than only one item. Here, a natural approach is to run k parallel instances of the described procedure to get a random sample containing k items. As long as k is sublinear in m, the storage required for this method is sublinear in m as well. However, it is important to note that such a procedure results in a random sampling *with replacement* where each item in the sample is chosen from the whole stream of m items. Hence, an item from the stream can be selected more than once into the sample.

In contrast, it is often useful—as we will later see in this section—to get a random sample *without replacement* where the sampled group of k items is randomly selected from all $\binom{m}{k}$ subsets of size k that are possible over a set of m items. To get such a sample, the approach of reservoir sampling can be generalized as follows.

The first k items $a_1, a_2, \dots, a_k$ in the stream are stored by the algorithm. For every subsequent item a_i, $k < i \le m$, the algorithm decides to include a_i into the sample with probability k/i. If a_i is chosen for insertion into the sample, the algorithm picks an item uniformly at random among the k stored sample items that is replaced by a_i.

This generalized reservoir sampling approach reads the stream sequentially and only stores k items from the stream, hence, provided k is sublinear in m, that yields a streaming algorithm. It also achieves the promised uniformity of the random sample. To see this, we consider the probability that any k-item subset from the stream is chosen as the final sample.

For the sake of simplicity and because for different sample sizes the reasoning is along the same lines, we focus on the case where $k = 2$, that is, a sample containing two items is desired. Let a_{ℓ_1}, a_{ℓ_2} be any two items from the stream where $\ell_1 < \ell_2$. To end up with those items as the final sample, a bunch of events must occur: First, the algorithm selects a_{ℓ_1} into the sample. Second, every a_i, $\ell_1 < i < \ell_2$ is either not chosen into the sample or, if it is chosen, it does not replace a_{ℓ_1}. Third, the algorithm selects a_{ℓ_2} into the sample but does not replace a_{ℓ_1} by doing so. Finally, no a_j, $\ell_2 < j \leq m$, is chosen into the sample. We combine the probabilities for these necessary events to get the probability for obtaining a_{ℓ_1} and a_{ℓ_2} as the final sample:

$$\begin{aligned}
Pr[\,a_{\ell_1}, a_{\ell_2} \text{ form the final sample}\,] &= \frac{2}{\ell_1} \cdot \prod_{\ell_1 < i < \ell_2} \left(1 - \frac{2}{i} + \frac{2}{i} \cdot \frac{1}{2}\right) \cdot \frac{2}{\ell_2} \cdot \frac{1}{2} \cdot \prod_{\ell_2 < j \leq m} \left(1 - \frac{2}{j}\right) \\
&= \frac{2}{\ell_1 \cdot \ell_2} \cdot \prod_{\ell_1 < i < \ell_2} \frac{i-1}{i} \cdot \prod_{\ell_2 < j \leq m} \frac{j-2}{j} \\
&= \frac{2}{(m-1) \cdot m} \\
&= 1/\binom{m}{2}\,.
\end{aligned}$$

It follows that all $\binom{m}{2}$ subsets of 2 items from the stream are equally likely to show up as the final sample.

We now want to direct our attention to an application for the reservoir sampling approach. Consider a situation where we see a stream of items $a_1, a_2, \ldots, a_m$ and after the stream we are given an order-independent predicate. Every item in the stream does or does not satisfy such a predicate, but that is independent of the item's position in the stream. The task is to report the number of items in the stream that satisfy the predicate. As an example we can imagine for our stream of natural numbers the predicate of being a prime number. Note that the catch here is that the predicate is announced *after* the items passed by. Hence, we cannot simply count the number of items satisfying the predicate while reading the input stream because the subsequent predicate might as well ask for the number of items being equal to zero or being larger than twelve; we simply do not know the predicate while receiving the stream.

This problem is called *query selectivity problem* and it is of importance in the area of databases. Here, a user's query is usually unknown while an update stream is fed into the database. To select a fast evaluation method for a query, it is very useful to know the fraction of tuples that are retrieved in each evaluation step, that is, the fraction of tuples that are selected by some predicate, c. f. [59].

It is not hard to imagine that any algorithm that exactly solves the query selectivity problem for general streams and predicates requires the storage of the whole input stream. To be prepared for every possible subsequent predicate, no input item can be abandoned.

However, often we do not need an exact answer; we are instead—as for the query evaluation planning in a database—satisfied with an estimated one. For this, we can use the described random sampling approach: We draw a sample of size k uniformly at random from the input stream and get m by simply counting the number of input items. After the predicate is known, we simply determine the number k^+ of items in our sample satisfying the predicate. The number of items in the whole input stream meeting the predicate we estimate as $(m \cdot k^+)/k$. That is, we calculate the fraction k^+/k of items in our sample satisfying the predicate and estimate that the same fraction of all m items in the stream meet the predicate as well.

Using the reservoir sampling approach, we can draw a sample of arbitrary size uniformly at random. On the one hand, we want k to be large since that clearly increases the accuracy of the estimate. On the other hand, k also determines the space consumption of the sampling algorithm since every sampled item must be memorized, thus, we want k to be small. Recall that k needs to be sublinear in m to give rise to a streaming algorithm. So, what size do we need for k?

Let m^+ be the number of input items that satisfy the given predicate, i. e., the number we want to estimate, and let $m^+ = m/c$ for a constant c, $0 < c \leq 1$. Assume that we aim for a $(1 \pm \varepsilon)$ - estimate of m^+ with probability $1 - \delta$ where ε and δ are constants with $0 < \varepsilon, \delta < 1$. That is, we want to have our estimate within the interval $[(1-\varepsilon)m^+, (1+\varepsilon)m^+]$ with probability $1 - \delta$. The parameters ε and δ are chosen by the user and affect the space consumption of the algorithm.

Since a sampled item satisfies the predicate with probability $1/c$, the expected value for k^+ is k/c. To achieve a $(1 \pm \varepsilon)$ - estimate of m^+ with probability $1 - \delta$, we need k^+ to be within the $(1 \pm \varepsilon)$ - interval around its expected value with the same probability. By an application of the Chernoff Bound, we have

$$Pr\Big[\,\big|k^+ - Exp[k^+]\big| \;>\; \varepsilon \cdot Exp[k^+]\,\Big] \;<\; e^{-\Theta(\varepsilon^2 \cdot k/c)}\;.$$

Consequently, if we draw $k = \mathcal{O}(1/\varepsilon^2 \log(1/\delta))$ samples from the input stream, the probability that k^+ is outside the $(1 \pm \varepsilon)$ - interval around its expected value and therefore the probability that we over- or underestimate m^+ by more than an ε - fraction is at most δ.

We emphasize the fact that a constant number of samples suffices, a number that is independent of the stream's length m. The same number of samples is sufficient to estimate the median or other quantiles from a stream [53].

Since a single item from U can be memorized in $\log n$ bits and the counter for m uses $\log m$ bits, the described approach requires a memory of $\mathcal{O}(\log n + \log m)$ bits. The input items are processed in a sequential fashion, thus, the whole approach is a streaming algorithm.

However, we do not want to conceal that the quadratic dependence of the sample size k on ε makes the scheme impractical for very small values of ε. For those cases more sophisticated sampling approaches are known, e. g. [54], that reduce the dependence on ε.

3.2 AMS-Sampling

We recall that every stream $a_1, a_2, \ldots, a_m$ describes a distribution on the universe U; it conveys information about the number of occurrences for every $j \in U$. As defined in Section 2, f_j denotes the number of occurrences of the element j in the input stream, that is, the frequency of j. If we imagine the stream $2, 3, 3, 2, 3, 1, 2, 3$ as an example, we get $f_1 = 1$, $f_2 = 3$, $f_3 = 4$, and $f_b = 0$ for every $b \in U \setminus \{1, 2, 3\}$.

There are many approaches in the area of streaming algorithms to reveal the characteristics of frequencies: The average, minimum/maximum values, the median and other quantiles can be estimated [62], as well as the most frequent items [25], the fraction of rare items [28], and histograms [35].

For each $k \geq 0$ we define $F_k = \sum_{j=1}^{n} f_j^k$ to be the kth *frequency moment.* Apart from F_1, which simply equals the length m of the stream, the frequency moments provide meaningful parameters of a distribution. F_0 gives the number of distinct items in the stream which can be used to detect denial of service attacks [8]; F_k for $k \geq 2$ characterizes the skew of the distribution and is used for example by query optimizers in databases when join sizes need to be predicted [5]. For our example stream, we have $F_2 = 26$ and $F_0 = 3$ where the second equation tells us that the stream consists of three different items.

It is easy to compute all frequency moments exactly by maintaining a counter f_j for every single item j of the universe. But of course, the memory consumption of such an approach heavily depends on the distribution and can be proportional to m and/or n; no streaming algorithm can emerge from this scheme. However, there is no other method, elaborated or not, that gives rise to a streaming algorithm since it is known [6] that every algorithm that exactly computes F_k for $k \neq 1$ requires storage linear in m and n.

Consequently, we have to be content with an approximative solution. As earlier for the query selectivity problem in Section 3.1, we aim for an $(1 \pm \varepsilon)$ - estimation of F_k with probability $1 - \delta$. More formally, we want to have a solution that lies within the interval $[(1-\varepsilon)F_k, (1+\varepsilon)F_k]$ with probability $1 - \delta$. Again, it is the users choice to set the constants $0 < \varepsilon, \delta < 1$ affecting the precision and memory usage of the algorithm.

A method to estimate frequency moments in the desired accuracy using only sublinear space is the procedure of *AMS-sampling.* It originates from a celebrated paper [6] of Alon, Matias, and Szegedy, hence the name. The method works for all F_k with constant $k \geq 1$ and is a sample-and-count approach where a sample is maintained with additional data.

The core of the AMS-sampling is to pick an item a_i uniformly at random from all items $a_1, a_2, \ldots, a_m$ in the stream and to compute $r = |\{i' : i' \geq i, a_{i'} = a_i\}|$, i. e., the number of items equal to a_i occurring in the stream starting at position i. At the end of the stream a value X is calculated as $X = m(r^k - (r-1)^k)$.

We can use the reservoir sampling approach from Section 3.1 to select a_i. Whenever the reservoir sampling chooses a new item to be the actual sample s, a counter c is initialized to one; every subsequent item in the stream that is not chosen to be sampled increases c by one if it equals s. At the end of the stream, r is provided by c.

In order to compute X at the end of the stream, we need to store s, c, and a counter for m; for that, $\mathcal{O}(\log n + \log m)$ bits are sufficient which is also enough for the actual calculation of X. Since the input stream is processed sequentially, this gives rise to a streaming algorithm.

Let us assume that for our example stream $2, 3, 3, 2, 3, 1, 2, 3$ we have $k = 3$ and our randomly picked item is a_4, that is, the second occurrence of 2 in the stream. Since there are two items equal to a_4 occurring in the stream starting at a_4 (namely a_4 and a_7), it is $r = 2$. Using $m = 8$ as the length of the stream we get $X = 8(2^3 - 1^3) = 56$.

It is striking that the somewhat inscrutable value X in fact estimates the demanded F_k. Therefore, X is what we call an *unbiased estimator*, that is, a variable whose expectation equals—without any further transformations—the value in question. To see this, we compute the expected value of X. Let U' be the subset of the universe U containing the items that occur in the stream, i. e., $U' = U \cap \{a_1, a_2, \ldots, a_m\}$. For every item $j \in U'$, any of the f_j occurrences of j in the stream might be selected as the final sample of the reservoir sampling procedure. Thus, X takes the value $m((f_j - r + 1)^k - (f_j - r)^k)$ if the rth occurrence of the item j is chosen as the final sample. Every occurrence of every item in U' is selected as the final sample with uniform probability $1/m$, thus, the possible values of X corresponding to those selections emerge with the same uniform probability. For the expected value of X, that means

$$Exp[X] \;=\; \sum_{j\in U'} \sum_{r=1}^{f_j} \left(m\big((f_j - r + 1)^k - (f_j - r)^k\big) \cdot \frac{1}{m} \right) \;=\; \sum_{j\in U'} f_j^k \;=\; F_k\,.$$

Even though in expectation X equals the desired value, it is not enough to simply take X as an estimator for F_k. We cannot be sure that the probability of X lying outside the $(1 \pm \varepsilon)$ - interval around F_k is at most δ as demanded. Therefore, the probability that X lies within this interval needs to be increased, that is, we have to boost the concentration of X around its expectation. To this aim the authors of [6] make use of a technique that has become a standard by now and is presented in the following.

To increase the concentration of the random variable X around $Exp[X]$, the variance of X—which is a measure for expected deviation of X from $Exp[X]$—needs to be reduced. This can be done for independent and identically distributed random variables $v_1, v_2, \ldots, v_s$ by taking the average $v^* = \sum_{a=1}^{s} v_a/s$ of the individual v_a's. We then have $Var[v^*] = Var[v_a]/s$, that is, compared to a single random variable v_a, the variance of the average is reduced by a factor of s. Note that the expected value of the average is the same as for each individual random variable v_a.

Now the idea to enhance the quality of the estimator for F_k is obvious: Instead of taking a single X as an estimator for F_k, we run s_1 independent instances of the above approach for X in parallel to compute values $X_1, X_2, \ldots, X_{s_1}$. By taking the average Y of this values, we get an estimator for F_k which is more concentrated around its expectation, that is, around F_k, than any individual X_a, $1 \le a \le s_1$.

To get the number s_1 of parallel copies that are required, we utilize Chebyshev's inequality and can deduce that

$$Pr\big[\,|Y - F_k| \;>\; \varepsilon \cdot F_k\,\big] \;\le\; \frac{Var[\,Y\,]}{\varepsilon^2 \cdot F_k^2} \;=\; \frac{Var[\,X_a\,]}{\varepsilon^2 \cdot F_k^2 \cdot s_1} \qquad \text{for all } 1 \le a \le s_1\,.$$

The authors of [6] show that for all $1 \le a \le s_1$, it is $Var[X_a] \le kn^{1-1/k}F_k^2$. Thus, by choosing s_1 to be $8kn^{1-1/k}/\varepsilon^2$, we get the following inequality[1]:

$$Pr\big[\,|Y - F_k| \;>\; \varepsilon \cdot F_k\,\big] \;\le\; \frac{Var[\,X_a\,]}{\varepsilon^2 \cdot F_k^2 \cdot s_1} \;\le\; \frac{kn^{1-1/k}F_k^2}{\varepsilon^2 \cdot F_k^2 \cdot 8kn^{1-1/k}/\varepsilon^2} \;=\; \frac{1}{8}\,. \tag{1}$$

Thus, the probability that Y is not a $(1 \pm \varepsilon)$ - estimation of F_k is at most $1/8$. Admittedly, we want this failure probability to be δ, not $1/8$. Of course, we could choose s_1 to be $kn^{1-1/k}/\varepsilon^2\delta$ instead of $8kn^{1-1/k}/\varepsilon^2$ to reduce the probability in (1) to δ. By doing so, s_1 would depend proportionally on $1/\delta$. But we recall that s_1 determines the memory consumption of the algorithm since every of the s_1 parallel instances of the sample-and-count approach needs to individually memorize a sample and a counter. Thus, a proportional dependence of s_1 on $1/\delta$ yields a handicap for applications where a very small failure probability is desired.

Fortunately, we can do better. Instead of taking a single Y as an estimator for F_k, we independently compute s_2 such values $Y_1, Y_2, \ldots, Y_{s_2}$ and take its median Z as our estimator for F_k. Every Y_b, $1 \le b \le s_2$ is the average of a separate group of s_1 X_a's as described above.

Let $Y^- = \{b : Y_b \notin [(1-\varepsilon)F_k, (1+\varepsilon)F_k],\ 1 \le b \le s_2\}$ be the set of indices of those Y_b's that are not an $(1 \pm \varepsilon)$ - estimate of F_k. Because of (1), we know that in expectation these

[1] The constant 8 in the value of s_1 is used in the original work [6]; any constant greater than two suffices and only slightly changes the line of argumentation in the following.

Y_b's are at most a 1/8 - fraction of all Y_b's, thus, $Exp\big[|Y^-|\big] \leq s_2/8$. If Z as the median of all Y_b's is no $(1 \pm \varepsilon)$ - estimate of F_k, it must hold that at least half of all Y_b's are no $(1 \pm \varepsilon)$ - estimate of F_k either. That only happens if the size of Y^- exceeds $s_2/2$, that is, it exceeds its expected value by at least $3s_2/8$. As a result, we can bound the failure probability for Z as

$$\begin{aligned} Pr\Big[Z \notin [(1-\varepsilon)F_k, (1+\varepsilon)F_k]\Big] &\leq Pr\Big[|Y^-| \geq Exp\big[|Y^-|\big] + 3s_2/8\Big] \\ &< e^{-\Theta(s_2)} \end{aligned}$$

where the second inequality follows from an application of the Chernoff bound. If we choose s_2 to be $\mathcal{O}(\log(1/\delta))$, the probability of Z being no $(1 \pm \varepsilon)$ - estimate of F_k is at most δ.

Altogether, our estimator for F_k is the value Z which is the median over s_2 independent Y_b's where each of those is the average of s_1 independent X_a's. Since every X_a requires the storage of a sampled item and a counter, the overall space requirement is $s_1 \cdot s_2 \cdot \mathcal{O}(\log m + \log n)$ bits which is $\mathcal{O}(kn^{1-1/k}(\log m + \log n)\log(1/\delta)/\varepsilon^2)$. Clearly, the presented scheme is a streaming algorithm as the memory consumption is sublinear in both m and n; additionally, a sequential access to the input stream suffices to realize the sample-and-count approach for the individual X_a's.

The presented achievement has been the foundation for a lot of work enhancing it. The biggest improvement is the reduction of the $n^{1-1/k}$ - factor in the presented space bound to an $n^{1-2/k}$ - factor [44]. This dependency on n is optimal, that is, cannot be decreased any further [11]. The dependency on ε^2 is impossible to reduce either [65].

All mentioned results hold for the $(1 \pm \varepsilon)$ - estimation with probability $1 - \delta$ of F_k for general k. It is interesting to note that if k is an integer with $0 \leq k \leq 2$, F_k can be estimated by a streaming algorithm using only $\mathcal{O}((\log m + \log n)\log(1/\delta)/\varepsilon^2)$ bits of memory [6]. For the special case of F_0, that is, the determination of the number of distinct elements in a stream, the work of Kane et al. [47] gives a space optimal algorithm.

3.3 Sliding Window Sampling

So far, we viewed all items in the input stream as being equally important, no matter how far the occurrence of an item dates back. That is fine for many applications; the design of a query evaluation plan in a database often is independent of the input item's chronological order. We have seen in the previous sections that the estimation of query selectivities or join sizes uses samples that are uniformly drawn from the whole input.

However, it is easy to come up with applications where recent items are more significant than older ones. A typical task is the prediction of a system's behavior in the future based on its current state; to determine the current state, the recent input is of importance. For instance, the Random Early Detection protocol RED [34] is used within Internet routers to anticipate traffic bottlenecks by maintaining statistics over the recent queue lengths. Another example is to track calling patterns of phone company customers. To identify rapid changes in calling behavior, companies keep track of weighted averages where recent behavior is given a larger weight than older one [27].

The easiest way to model a different influence of older and newer input items is to simply define a stream's location ℓ such that all stream items a_i with $i \geq \ell$ are viewed as equally significant while all items a_i with $i < \ell$ are not considered at all. If we are interested in the recent w items, we have to increment ℓ for every incoming item. This approach is called *sliding window model*. Formally, instead of looking at the whole input stream $a_1, a_2, \ldots, a_t$, where a_t is the last item arrived, we only look at the items $a_{t-w+1}, a_{t-w+2}, \ldots, a_t$. We can visualize this scheme as a window of size w that slides over the input stream from left to right, hence the name.

There has been work in the sliding window model tackling problems that are known from the ordinary data stream model. In [23] different statistics and histograms of a sliding window are computed; [28] shows how to estimate the fraction of rare items and the similarity of different streams. We will take a look at such applications in Section 5.2. In the present section, we want to focus on the problem of maintaining a sample from the sliding window, a problem which serves as a foundation for more evolved procedures.

The memory consumption of a streaming algorithm is constrained to be sublinear in m and n. Note that under this requirement, every algorithm that completely stores the content of the sliding window is a streaming algorithm as long as the window size w is sublinear in m and n. Since for larger w such an algorithm is unsuitable, we have to tighten our requirement accordingly. To yield practicable algorithms, we demand the memory usage of a sliding window algorithm to be sublinear in n and w. As well as in the ordinary streaming model, for most functions, their exact computation is impossible in this model [29] and we strive for approximative solutions.

We want to give a sliding window algorithm that maintains a uniform random sample of all items contained in the actual window. As in Section 3.1, we emphasize on the fact that the sample is drawn uniformly from all items, that is, all positions within the window, not from the universe elements that are part of the window. For the sake of simplicity, we aim for a sample of size one and comment on larger sample sizes at the end of this section.

It is obvious that the reservoir sampling approach alone is not sufficient for the sliding window context. Of course, we might be lucky and every actual sample arises from the actual sliding window. But it is more likely that the actual sample falls out of the window as it progresses; at that moment any algorithm using memory sublinear in w cannot construct a new uniform sample.

A tempting idea to overcome this might be the following: We use the reservoir sampling approach only over the first w stream items to draw a sample a_ℓ. By the properties of the reservoir sampling, for the first sliding window $a_1, a_2, \ldots, a_w$ the item a_ℓ is chosen uniformly at random. If in the following the sliding window moves on, we keep a_ℓ as the random sample until it falls out of the window, that is, if the item $a_{\ell+w}$ appears. At that particular moment, we take $a_{\ell+w}$ as our sampled item which in turn is replaced with $a_{\ell+2w}$ and so on. That way, the item $a_{\ell+c\cdot w}$, $c \in \mathbb{N}$, is the actual random sample as soon as it occurs in the stream. If we look at each sliding window individually, this approach indeed yields a sample that is drawn uniformly at random from the window content. However, the sample for different window positions is profoundly dependent: The place of a random sample for one window position completely determines the place of the random sample in all following window positions. Clearly, that is infeasible for many applications.

There is an algorithm for sampling items uniformly at random from a sliding window by Braverman et al. [12]. In the following, we want to investigate the simple sliding window algorithm maintaining a uniform random sample that is proposed by Babcock et al. [10] and uses a *priority sampling* approach. Every incoming item a_i is given a priority $p(a_i)$, that is, a random value chosen uniformly at random in the continuous interval between 0 and 1. The actual sample a_ℓ is given by the item that has the highest priority among all items in the sliding window.

It remains to see that this method can be realized within the memory constraints of the sliding window model. To this aim, we note that it is not necessary to store all items from the actual window. We only need to memorize those items whose priority is maximal among items that arrived later. This is because an item a_r will never be the sample if there is an item a_s with $s > r$ such that $p(a_s) > p(a_r)$. Hence, a_r can be abandoned. We use a linked list L that is ordered by decreasing priority to store the items that could be the actual sample in the future. Every input item a_i is processed by the random drawing of its $p(a_i)$, its insertion into L according to $p(a_i)$, and the deletion of all descendants of a_i in L. None of these descendants can become the sample anymore. Furthermore, the item a_{i-w}, that is, the item that fell out of the window on the arrival of a_i, is erased from L if it is part of L. Note that at any time, the actual random sample is given by the head of the linked list.

Let us examine the key question here: What is the length $|L|$ of the linked list L, i. e., how many items do we need to store? Clearly, the constitution of L depends on the item's priorities and so does $|L|$. It is interesting that in fact $|L|$—and therefore the memory consumption of the algorithm—is a random variable. That includes the chance that the random choices of the priorities force the memorization of all window items in which case $|L| = w$ and the model's memory constraint is violated. However, we will see that such a violation is very unlikely by an argumentation that is inspired by [7].

To this aim, we calculate the expected value of L's length, i. e., $Exp[|L|]$. Let $a'_1, a'_2, \dots, a'_w$ be the w recent input items, that is, the items that are in the actual window where a'_i arrived before a'_{i+1}. Since a'_w becomes the end of L, $|L|$ is given by the number of ancestors of a'_w in L (where we define every item in L to be an ancestor of itself). Let $X_1, X_2, \dots, X_w$ be indicator random variables such that $X_i = 1$ if a'_i is an ancestor of a'_w in L and $X_i = 0$ otherwise. The crucial observation is that a'_i is an ancestor of a'_w iff among all a'_h with $i \le h \le w$ the priority $p(a'_i)$ is maximal. Since for s independent identically distributed continuous random variables a fixed variable takes the maximum with probability $1/s$, we have $Pr[X_i = 1] = 1/(w-i+1)$. The X_i's are 0-1-random variables, thus $Exp[X_i] = Pr[X_i = 1]$. Due to the fact that a'_w cannot have ancestors with a later arrival time and by the linearity of expectation,

$$Exp\big[|L|\big] = Exp\left[\sum_{i=1}^{w} X_i\right] = \sum_{i=1}^{w} Exp\big[X_i\big] = \sum_{i=1}^{w} \frac{1}{w-i+1} = H_w$$

where H_w is the wth harmonic number bounded by $\ln w < H_w \le \ln w + 1$. Hence, in expectation, only a logarithmic number of items is stored in the linked list L. To see that with high probability no significant deviation from this expectation occurs, we apply the Chernoff bound on $|L|$ as a sum of indicator random variables having different distributions. According to this, for every constant $c \ge e^2$,

$$Pr\Big[\,|L| \ge c \cdot Exp\big[|L|\big]\Big] < e^{-c \cdot Exp[|L|]} = e^{-c \cdot H_w} < w^{-c}$$

which means that with high probability the length of the linked list is $\mathcal{O}(\log w)$.

It is important to note that for the above analysis we have to assume that all priorities are distinct. From a theoretical point of view that is no issue since two samples from a continuous interval differ with probability one. But a streaming algorithm using limited storage cannot memorize arbitrary real values from a continuous interval. To overcome this, we use a technique of [7]: The random priorities in the interval $[0, 1]$ are generated piecemeal by adding more and more random bits to their binary representation when required. We only use the priorities for comparisons; if for two compared priorities one binary representation is the prefix of the other, the representations are randomly enhanced until the comparison is decided. By [7], with high probability it suffices to generate—and memorize—only a constant number of bits for every priority.

After all, the described technique yields a streaming algorithm to draw from a sliding window of size w a sample that is uniformly distributed over all items in the window. The input items can be processed in any given order. Since with high probability $\mathcal{O}(\log w)$ items are memorized in the linked list and every memorized item requires $\mathcal{O}(\log n)$ bits of storage, the memory consumption is $\mathcal{O}(\log w \cdot \log n)$ with high probability.

It remains to enhance the algorithm for drawing more than one sample. As mentioned in Section 3.1, the execution of k parallel independent runs yields a sample of size k which is drawn with replacement. We can simulate sampling without replacement by executing additional independent runs to achieve at least k distinct samples among all samples with high probability. As long as k is sublinear in w, the number of required additional runs is sublinear as well [10].

4 Sketching

The challenge for a streaming algorithm is to make a space-efficient summarization of the input that allows to answer the given query at the end of the stream. In the previous section we have examined the summarization due to sampling which is space-efficient as only a limited number of input items are memorized. In this section a different technique called sketching is considered.

Recall that for the cash register model we assumed the input stream $a_1, a_2, \dots, a_m$ to be a sequence of items where each a_i stems from a universe U of size n. We may regard this stream as an implicit, incremental update to a vector $f = (f_1, f_2, \dots, f_n)$ of dimension n. Initially, f is the zero vector, i. e., $f_j = 0$ for all $1 \leq j \leq n$. Each input item a_i in the stream updates f by incrementing f_{a_i} by one and leaving all other vector entries untouched. Hence, after reading input item a_t, $1 \leq t \leq m$, the vector f is the *frequency vector* of the stream $a_1, a_2, \dots, a_t$, that is, each vector entry f_j equals the number of occurrences of element $j \in U$ in $a_1, a_2, \dots, a_t$ as previously defined in Section 2.

In the more general turnstile model, every item in the stream is a pair $(j, z) \in U \times \mathbb{Z}$. We can imagine that after each such pair (j, z), the frequency vector f is updated by adding z to f_j. Recall that a positive z corresponds to insertions of item j, a negative z to deletions. While in the strict turnstile model we assume all vector entries f_j of f to be non-negative at all times, in the non-strict case, the f_j's can be general values in $\mathbb{Z}$. In both cases, the cardinality m of the stream is given by the sum of the absolute values of all vector entries.

Since in the streaming context we cannot assume or exclude particular orders of the input items, every reasonable function to be calculated by a streaming algorithm is order-invariant. Note that such an order-invariant function on the input items could easily be computed using the frequency vector f at the end of the stream. Admittedly, no streaming algorithm can have f at its disposal because the memorization of a general n-dimensional vector requires at least n bits which violates the memory constraints of the streaming model. However, we could, in limited space, try to sketch f in a way that allows the approximation of the demanded function at the end of the stream.

That is exactly what the *sketching* approach does. It uses pseudo-random vectors of dimension n and computes the dot product of these vectors with the frequency vector f. In particular, if x is a pseudo-random n-dimensional vector, the dot product $f \cdot x^T$ is called a *sketch* of f. Usually, several sketches are used in combination to compute a—naturally randomized—approximation of the function in question.

There are two important reasons for the utilization of sketches in the streaming context: First, the sketch of the frequency vector f can be computed gradually while the stream items, that is, the incremental updates of f are processed. For every input item $(j, z) \in U \times \mathbb{Z}$, the sketch needs to be increased by $z \cdot x_i$ where x_i is the ith entry of x. Second, the sketch can be maintained in small memory. Since we assume the entries of x to be constants, the size of the sketch $f \cdot x^T$ is $\mathcal{O}(m \cdot n)$ which can be memorized in $\mathcal{O}(\log m + \log n)$ bits.

One of the first utilizations of sketches for the streaming context can be found in the seminal paper of Alon, Matias, and Szegedy [6]. Here, the authors improve their own result of approximating F_2—which we examined in Section 3.2—by exploiting sketching techniques. A further example of a sketching algorithm is the estimation of the number of distinct elements in a data stream [24].

4.1 Count-Min Sketch

To highlight the sketching approach's efficacy, we want to tackle the *point query* problem. It asks at the end of the stream for f_j, i. e., the number of occurrences in the input of an arbitrary item $j \in U$. Note that the problem corresponds to the index problem of communication complexity [50] which means that the storage required for an exact answer is $\Omega(n)$. That is not surprising; since the algorithm does not know j before the end of the stream, it has to prepare itself for every possible point query which for general streams implies to maintain a counter for every item of the universe. It is further known [50] that even a randomized algorithm with a reasonable error probability for this problem must use a memory of size linear in n.

Hence, any streaming algorithm must be satisfied with an approximative answer. We present such an algorithm in the following that utilizes the sketching approach, in particular, the *count-min sketch* proposed in [26]. Our aim is to answer any point query by giving an estimate $\widehat{f_j}$ of the queried f_j. For this estimate we demand that $f_j \leq \widehat{f_j} \leq f_j + \varepsilon \cdot m$ with probability $1 - \delta$. As usual, the constants $0 < \varepsilon$ and $0 < \delta < 1$ are selected by the user in a trade-off between desired precision and memory consumption of the algorithm.

For the ease of presentation, we focus in the following on our canonical cash register input stream $a_1, a_2, \ldots, a_m$ where each a_i is part of the universe U.[2] However, we emphasize that the presented count-min sketch smoothly applies to the strict turnstile model.

We let $w = \lceil 2/\varepsilon \rceil$ and set up an array of w counters $c(1), c(2), \ldots, c(w)$ initialized with zero. From a family of 2-universal hash functions that map from U to $\{1, 2, \ldots, w\}$, a function h is chosen uniformly at random. We comment on the usage and shape of such hash functions in Section 4.2. While reading the input stream, the algorithm updates the counters: For every input item a_i, the counter $c(h(a_i))$ is incremented by one. After reading the whole stream, the algorithm answers any point query by utilizing the counters. In particular, if f_j is in question, the algorithm provides $f'_j = c(h(j))$ as an estimate.

It is easy to see that $f'_j \geq f_j$ because every single occurrence of j in the stream increases the counter $c(h(j))$. However, we cannot expect $f'_j = f_j$ because the number of counters is smaller than the size of U, thus, collisions occur which affect the counter for item j to be counted over with different input items. Let us examine the key question of how many excessive increasings we have to reckon with.

For a fixed point query on f_j, let $X_1, X_2, \ldots, X_n$ be indicator random variables such that $X_i = 1$ if $h(i) = h(j)$ and $i \neq j$; otherwise $X_i = 0$. Intuitively, each X_i indicates if an item i different from j is hashed to the same counter by h. Since for the 0-1-variables

[2] Notice that this cash register stream corresponds to the stream $(a_1, 1), (a_2, 1), \ldots, (a_m, 1)$ in the turnstile model.

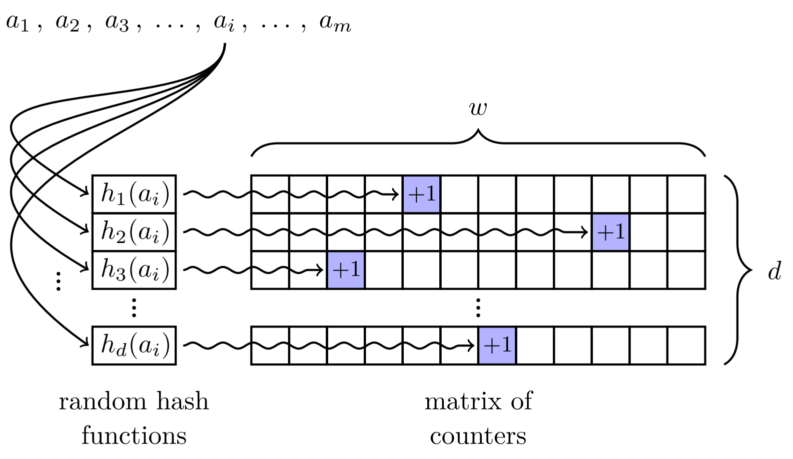

Figure 2 Update of the count-min sketch for input item a_i of the input stream. For each row $1 \le s \le d$, the assigned hash function h_s is evaluated for a_i. The result $h_s(a_i)$ indicates which column to increment in row s.

$Exp[X_i] = Pr[X_i = 1]$, we get

$$Exp[X_i] \;=\; Pr[\,h(i) = h(j)\,] \;=\; \frac{1}{w} \quad \text{for } i \in U\backslash\{j\} \text{ and } Exp[X_j] \;=\; 0 \tag{2}$$

where the first equality follows by the property of a function chosen randomly from a 2-universal family, see Section 4.2. Furthermore, we define $Y = \sum_{i=1}^{n} f_i X_i$ to be number of increasings to $c(h(j))$ that do not originate from j; hence, $f'_j = f_j + Y$. By linearity of expectation,

$$Exp[Y] \;=\; Exp\left[\sum_{i=1}^{n} f_i \cdot X_i\right] \;=\; \sum_{i=1}^{n} f_i \cdot Exp[X_i] \;\le\; \frac{m}{w} \;\le\; \frac{\varepsilon \cdot m}{2}\,.$$

Thus, in expectation the estimate f'_j exceeds the true value f_j by an amount of $\varepsilon \cdot m/2$. Using Markov's inequality, we can bound the probability that f'_j overruns f_j by a value greater than $\varepsilon \cdot m$ as

$$Pr[\,f'_j \;>\; f_j + \varepsilon \cdot m\,] \;=\; Pr[\,Y \;>\; \varepsilon \cdot m\,] \;\le\; \frac{Exp[Y]}{\varepsilon \cdot m} \;=\; \frac{1}{2}\,. \tag{3}$$

To reduce this failure probability to the desired value δ, we run $d = \lceil \log(1/\delta) \rceil$ independent runs of the described algorithm in parallel. We can imagine this scheme as a $d \times w$ matrix of counters $c(s,t)$ with $1 \le s \le d$, $1 \le t \le w$ where each row has its own hash function h_s. It is important that each of those functions is chosen independently and uniformly at random from a family of 2-universal hash-functions. Every input item a_i causes an update in every row, i. e., for every $1 \le s \le d$, the counter $c(s, h_s(a_i))$ is incremented by one. The final estimate $\widehat{f_j}$ for f_j is the minimum value over the row's estimates, that is, $\widehat{f_j} = \min\{c(s, h_s(j)) : 1 \le s \le d\}$.

This whole scheme is called count-min sketch [26] based on its two main operations counting and minimizing. Figure 2 shows the update of the sketch for an input item a_i; the determination of the returned value $\widehat{f_j}$ is outlined in Figure 3.

It remains to certify the claimed quality of the count-min sketch's estimate. To bound the failure probability of providing an estimate $\widehat{f_j}$ that exceeds the true value f_j by more than

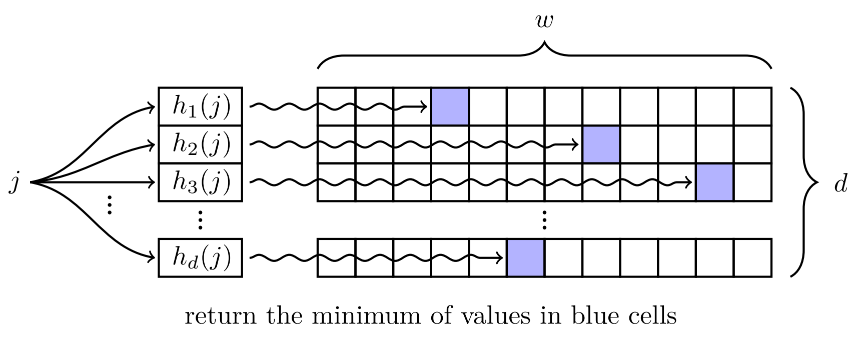

Figure 3 Count min sketch estimation of f_j. For each row $1 \leq s \leq d$, the assigned hash function h_s is evaluated for j. The result $h_s(j)$ denotes the column to consider in row s. Of all considered entries—indicated blue in this figure—the minimum is returned as $\widehat{f_j}$, that is, the estimation of f_j.

$\varepsilon \cdot m$, note that this happens iff the estimates of all rows overrun f_j by more than $\varepsilon \cdot m$ as well:

$$Pr[\widehat{f_j} > f_j + \varepsilon \cdot m] = Pr[\text{for all } s \in \{1, \dots, d\} : c(s, h_s(j)) > f_j + \varepsilon \cdot m] \leq 2^{-d} \leq \delta$$

where the first and second inequality is due to (3) and the choice of d, respectively. Thus, the failure probability is as desired.

Let us finally see that this procedure indeed yields a streaming algorithm. Obviously, the input stream is processed sequentially. For the memory usage of the algorithm we state that the hash functions consume a space of $\mathcal{O}(\log n \cdot \log(1/\delta))$; we postpone the reason for that to the next section. Any of the $d \cdot w$ counters can hold a value of at most m which yields an overall memory usage of $\mathcal{O}(\log m \cdot \log(1/\delta)/\varepsilon + \log n \cdot \log(1/\delta))$ bits satisfying the limits of the streaming model.

The utilization of the count-min sketch in the strict turnstile model instead of the cash register one is straightforward: For each stream item $(j, z) \in U \times \mathbb{Z}$, we add z to all counters that keep track of the occurrences of j. The estimation procedure for a query remains the same, as well as the answer guarantees and the memory consumption. However, for the non-strict turnstile model, the count-min sketch loses its ability to give an estimate $\widehat{f_j}$ that is an upper bound of f_j. This is due to the fact that the counters corresponding to j might underrun f_j because of colliding items with negative frequencies.

Finally, we want to highlight that the count-min sketch exploits its strength especially on a stream in the strict turnstile model. We can imagine such a stream as insert or delete operations to a database. As mentioned, with probability $1 - \delta$ the error of the sketch is $\varepsilon \cdot m$ where m now is the number of items currently in the database. As an example assume $\varepsilon = 0.1$ and $\delta = 0.01$, thus, the count-min sketch comprises $w \cdot d = 20 \cdot 7 = 140$ counters. We use it to track the insertion of a multiset containing a million ($< 2^{20}$) IP addresses into a database. Since each counter requires at most 20 bits, an overall of at most 2800 bits are used by the counters which is smaller than the input size by several orders of magnitudes. If now all but a multiset of nine addresses are deleted again, we can use point queries to the count-min sketch to reveal each of the remaining addresses and their frequencies exactly with a probability of 0.99 because with this probability the error for a point query is at most $9\varepsilon < 1$.

4.2 Universal Hash Functions

The sketching technique relies on projecting the frequency vector f along pseudo-random vectors to reduce the dimension of information to memorize. Often, this pseudo-random vectors are given implicitly by pseudo-random functions. We can see this in the case of the count-min sketch of the Section 4.1 where every single counter is a sketch. Here, a counter can be regarded as a dot product of f with a 0-1-vector that has a 1 at position i iff the hash function of the counter's row maps item $i \in U$ to the counter.

The reason for utilizing pseudo-random functions instead of completely random ones is the memory constraint of the streaming model; the memorization of completely random functions whose domain is the universe U is infeasible. This is due to the fact that in order to store a completely random function over the domain U we have to be prepared to store any function over U. That however requires the potential to memorize for each element in U which element of the target set is assigned to it; a memory of size $\Omega(|U|)$ is needed to do so.

In contrast, a suitable pseudo-random function combines some random-like properties with small required storage space. A basic family of such functions is the family of 2-universal hash functions.

As usual, $U = \{1, 2, \dots, n\}$ is our universe and let $V = \{0, 1, 2, \dots, q-1\}$ be a set with $q \leq n$. A family of hash functions $\mathcal{H}$ from U to V is said to be *2-universal* if, for all $x_1, x_2 \in U$ with $x_1 \neq x_2$, and for h chosen uniformly at random from $\mathcal{H}$ we have

$$Pr[\, h(x_1) = h(x_2) \,] \;\leq\; \frac{1}{q}\,. \tag{4}$$

This property reflects what we mean by a random-like behavior. It is something we expect a function to have that maps completely random from U to V, that is, assigns a completely random hash value to every item in U. Notice that the family of all functions from U to V satisfies this property as the random choice of any function from this family corresponds to a completely random mapping. However, since there are $|V|^{|U|} = q^n$ functions in this family, $\Omega(n \cdot \log q)$ bits are required to store such a function distinguishable from all others which exceeds the memory limitation.

However, there are families of functions that are 2-universal without being completely random. For a fixed prime $p > n$, we let $h_{a,b}(x) = (((ax + b) \bmod p) \bmod q)$ and define a family of hash functions as $\mathcal{H}' = \{h_{a,b} : 1 \leq a \leq p-1,\, 0 \leq b \leq p\}$. Each function h from this family is far from being completely random: The knowledge of two mappings $h(x_1)$ and $h(x_2)$ for $x_1 \neq x_2$ suffices to deduce $h(y)$ for every $y \in U$ while for a completely random function we can never deduce an unknown mapping from known ones. Anyway, $\mathcal{H}'$ can be shown to be 2-universal [55].

The crucial observation is that to memorize a function from $\mathcal{H}'$, we only need to store a, b, and p which can be done in $\mathcal{O}(\log n)$ bits[3]. Since the property of a 2-universal family is exactly what is needed for equality (2), we can utilize $\mathcal{H}'$ as the family of hash functions for the count-min sketch. For every row of the matrix of counters, a random function $h_{a,b}$ from $\mathcal{H}'$ is drawn by randomly selecting a and b within their respective bounds. For $\lceil \log(1/\delta) \rceil$ rows, $\mathcal{O}(\log n \cdot \log(1/\delta))$ bits are used to store the required hash functions of the count-min sketch.

In the original work presenting the count-min sketch [26], the authors choose the hash functions out of a family that is pairwise independent or strongly 2-universal. A family of

[3] Notice that by Bertrand's postulate, there is a prime p with $n < p < 2n$.

hash functions $\mathcal{H}$ from U to V is said to be *strongly 2-universal* or *pairwise independent* if, for all $x_1, x_2 \in U$ with $x_1 \neq x_2$, any $y_1, y_2 \in V$, and for h chosen uniformly at random from $\mathcal{H}$ we have

$$Pr[\,h(x_1) = y_1 \text{ and } h(x_2) = y_2\,] \;=\; \frac{1}{q^2}\,. \tag{5}$$

Note that this guarantee of pairwise independence between $h(x_1)$ and $h(x_2)$ is stronger than the one for the 2-universal family as property (4) follows from property (5). Even if the count-min sketch does not require this stronger guarantee, there are techniques used in the streaming area that do so or demand even stronger properties. The estimation of F_2 in [6] utilizes a family of strongly 4-universal hash functions where a family is strongly k-universal or k-wise independent if the hash values $h(x_1), h(x_2), \ldots, h(x_k)$ are mutually independent for all distinct $x_1, x_2, \ldots, x_k \in U$. For any constant k, there are constructions known [64] that yield a family of strongly k-universal hash functions from U to V where every function can be memorized using $\mathcal{O}(k \cdot \log n)$ bits. These small memory requirements make those functions a valuable tool for many streaming algorithms.

5 Similarity Mining

Estimating the similarity between two data streams is a basic problem in the data stream model and has many applications in mining massive streams of data, tracking changes in the network traffic, processing genetic data and query optimization. As an example, consider the problem of identifying similar entities (eg., web sites) based on the similarity between their corresponding data stream logs (IP addresses of their visitors, click-stream patterns, etc.).

An obvious solution to the problem of similarity estimation is to maintain a counter for each distinct item from the stream and compute the similarity at query time. Unfortunately, this solution requires $\Theta(n)$ words of storage, where n is the size of the universe U. As discussed previously, in the data streams scenario the dimensionality of the universe is typically very high, as well as the number of streams being analyzed. Very often we will not be able to afford the amount of memory which will be necessary in order to obtain exact answers. In such situations, one must refer to algorithms which will use bounded small amount of memory (polylogarithmic in the size of the universe), and will be able to produce high-quality approximations with high probability.

Among the most commonly used measures of similarity are the L_p distance and the Jaccard coefficient of similarity. In this section we will discuss algorithms for estimating these measures of similarity both in the *unbounded* data stream models (*time series model*, *cash register model* and *turnstile model*) and in the *windowed combinatorial* data stream model. The described algorithms build upon the basic mathematical ideas described in Section 3.3 and Section 4.

5.1 Estimating Similarity on Unbounded Data Streams

Random projections are an important mathematical idea used typically for an efficient dimensionality reduction over high cardinality domains. To this end many techniques have been proposed for computing various types of sketches, which rely on pseudo-random vectors generated by space-efficient computation of pseudo-random variables. The AMS-sampling and the count-min sketch described in Section 3.2 and Section 4 respectively are both based on the same general idea.

The idea of using multiple random projections is very general and works in the turnstile model as well. Similar powerful concept is based on generating a sequence of random variables each drawn from a stable distribution. Sketches based on different stable distributions are useful for estimating various L_p norms on the data stream, and form the basis of the algorithms presented in this section.

Another very useful mathematical tool is the family of min-wise hash functions whose properties enable a simple but efficient estimation of the Jaccard coefficient of similarity. Min-wise hashing has been used to estimate the similarity between two data sets representing various items in a market-basket analysis [22], and for estimating rarity and similarity in the *sliding window* or *combinatorial* data stream model [28].

5.1.1 L_2 and L_p Sketches

One of the most commonly used measures for data stream similarity is the L_p distance between two streams $A = (a_1, a_2, \ldots, a_m)$ and $B = (b_1, b_2, \ldots, b_m)$, where m is the length of the stream, while a_i and b_i are the actual i-th data elements of A and B. Here we consider the simplest *time-series* data stream model. For a real number $p \geq 1$ the L_p distance is defined by:

$$L_p = \sum_{i=1}^{m} |a_i^p - b_i^p|^{1/p}. \tag{6}$$

The the same definition applies to the *cash-register* and the *turnstile* data stream models with the difference that a_i and b_i would now represent updates on the counts of the corresponding stream elements $A[j]$ and $B[j]$, for $j \in \{1, .., n\}$.

The special cases of the L_p distance for $p = 0$ and $p \to \infty$ are defined as follows. The L_0 distance (also known as the Hamming distance) is the number of i's such that $a_i \neq b_i$, and measures the dissimilarity between two data streams. The L_∞ distance is the limit of L_p for $p \to \infty$ and is equivalent to the maximal difference at any time between any two items for the given data streams:

$$L_\infty = max_{i \in \{1,m\}} |a_i - b_i|. \tag{7}$$

There is a substantial amount of work done on estimating the L_1 [32, 24], the L_2 (Euclidean norm) [6, 43] and the L_p norm [43]. Feigenbaum et al. [32] were the first to produce a data stream algorithm for estimating the L_1 distance. Their technique relied on construction of pseudo-randomly generated "range-summable" variables which are *four-wise independent*[4]. The L_2 norm has been mostly used for estimating join and self-join sizes for the task of query selectivity estimation using only a limited storage. The earliest work for estimating the L_2 norm is the paper of Alon et al. [6], where they consider the simpler *cash-register* model. Their algorithms have been later extended in the work of [5] for handling the general sequence of insertions and deletions in the *turnstile* data stream model.

Alon et. al.'s technique for estimating the L_2 norm is based on the same concept described in Section 3.2. The main idea is to define a random variable which can be computed under the given space constraint, whose expected value is exactly the quantity we wish to estimate, and whose variance is relatively small. The final result is then obtained by considering

[4] The *four-wise independence* is defined as: the probability that a group of four random variables $\{\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4\}$ will map into a given combination of -1, +1 values, e.g., $\{-1, +1, +1, -1\}$, is equal to $1/2^4$=1/16.

sufficiently many such estimators, whose average is more concentrated around the expectation of a single estimator. By putting them into several groups, computing the average within each group, and taking the median of the group averages we get an estimator of the desired quantity whose variance is bounded by a user-defined parameter ε with a tunable probability of success 1 - δ, where δ is also specified by the user. Having the basic technique already described, here we will briefly outline only the main points of the algorithm.

Let $Z_{i,j}$ for $i = 1, 2, \ldots, s_1$ and $j = 1, 2, \ldots, s_2$ denote independent random variables defined as $Z_{i,j} = \sum_{v=1}^{m} \epsilon_v(a_v - b_v)$, where ϵ_v are 4-wise independent random variables that take on the values +1 or -1 with equal probability. Let $X_{i,j} = Z_{i,j}^2$. The interesting result is that, the expected value of the square of this quantity is the square of the L_2 distance we wish to estimate:

$$\begin{aligned} Exp[X_{i,j}] &= Exp\big[(\sum_{v=1}^{m} \epsilon_v(a_v - b_v))^2\big] \\ &= Exp\big[\sum_{v=1}^{m} \epsilon_v^2(a_v - b_v)^2 + \sum_{v \neq u} \epsilon_v\epsilon_u(a_v - b_v)(a_u - b_u)\big] \\ &= \sum_{v=1}^{m}(a_v - b_v)^2. \end{aligned}$$

The last equality follows from the fact that $Exp[\epsilon_v] = 0$ which will cancel out the second term, and $\epsilon_v^2 = 1$, as well as the independence of the random variables.

We now explain how to compute the variables $Z_{i,j}$, and hence $X_{i,j}$. Each $Z_{i,j}$ is initialized to 0. The resulting algorithm simply maintains the values of the random variables $Z_{i,j}$ after every update. Maintaining this value under continuous updates of the items a_v and b_v is straightforward: when an item with a value v arrives from stream A, we add ϵ_v to $Z_{i,j}$ for all i and j. If an item with a value v arrives from stream B, we subtract ϵ_v from $Z_{i,j}$ for all i and j. Practically, for each data item with value v we generate a mapping $h_{i,j}(v)$ from $\{1, 2, \ldots, m\}$ to $\{-1, +1\}$ which is being added/subtracted to the random variables $Z_{i,j}$.

The authors refer to this algorithm as *tug-of-war*, because each member of the sequence with a value mapping to +1 associates to pulling the rope in one direction, while each member with a value mapping to -1 associates to pulling the rope in the other direction. Note that this algorithm does not require a priori knowledge about the length of the sequence, or the number of distinct items seen from U at query time.

Let further Y_j be the average of $\{X_{1,j}, X_{2,j}, \ldots, X_{s_1,j}\}$ for all $j = 1, 2, \ldots, s_2$. Our final estimate is given with the value of Y which is the median of $\{Y_1, Y_2, \ldots Y_{s_2}\}$. As previously, parameter s_1 determines the accuracy of the results, i.e., the variance of the estimation, and parameter s_2 determines the confidence. By taking $s_1 = 1/\varepsilon^2$ and $s_2 = \log(1/\delta)$ the algorithm will need $\mathcal{O}(s_1 s_2) = \mathcal{O}(1/\varepsilon^2 \log(1/\delta))$ memory words.

Building upon the ideas in [6, 32], Indyk extended the previous results providing a unified framework for approximating the L_p distance between two data streams in small space, for any $p \in (0, 2]$ [43]. The method relies on the notion of *p-stable distributions.*

▶ **Definition 1.** A distribution D over $\Re$ is called *p-stable*, if there exist $p \geq 0$ such that for any n real numbers a_1, a_2, ..., a_n and i.i.d.[5] variables X_1, X_2, ..., X_n with distribution D, the random variable $\sum_i a_i X_i$ has the same distribution as the variable $(\sum_i |a_i|^p)^{1/p} X$ where X is a random variable with distribution D.

[5] independent and identically distributed

It is known that stable distributions exist for any $p \in (0, 2]$. In particular, the *Cauchy distribution* is 1-stable, and the *Gaussian (normal) distribution* is 2-stable, while for the general case $p > 2$, random variable X from a *p-stable* distribution can be generated by using the method of Chambers et al. [45].

The idea of using stable distributions enables us to use the previous approach for estimating the quantity $(\sum_i |a_i|^p)^{1/p}$ for any $p \geq 0$. The algorithm proceeds as previously by generating a number of i.i.d. random variables $X_{i,j}$, only this time drawn from a p-stable distribution D. The resulting random variables $Z_{i,j}$ will have "magnitudes" proportional to the "magnitudes" of the corresponding random variables $X_{i,j}$, which implies that the dot product can be used to approximate the value of the L_p distance. As previously one needs to repeat the procedure multiple times in parallel. The final estimate will be within a multiplicative factor $1 \pm \varepsilon$ of the true value, with probability of at least $1 - \delta$.

5.1.2 Min-wise Hashing

Another very popular measure of similarity is the Jaccard coefficient of similarity. Given two data streams A and B, let S_A denote the set of distinct items appearing in stream A, and let S_B denote the corresponding set for stream B. The Jaccard similarity between these two data streams is defined as:

$$\sigma(S_A, S_B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|}.$$

Since we are restricted on the space we can use the simplest approach of memorizing all the distinct items observed till the moment is not viable. Thus, we would have to do some sort of sampling, choosing not to memorize the appearance of some items. This equals to creating signatures of size $k \ll n$ bits for each set of distinct items, $Sig(S_A)$ and $Sig(S_B)$, which will be then used to compute an estimate of the similarity. Of course, the simplest way would be to sample the sets uniformly at random k times, using some of the techniques described above. However, due to sparsity this approach can miss important information, and as a result we would obtain a biased similarity estimate. This becomes obvious from the formula of the Jaccard coefficient, which shows that we are interested in the items that appear in both of the streams, while random sampling will not take into account this important fact.

Having in mind that streams are typically characterized with domains of a high cardinality, creating a space-efficient signature for each stream is not an easy task. Here we will present a very efficient way (in terms of memory and time) based on the concept of min-wise hashing [21]. Before explaining how it is possible to construct small signatures from large sets, it is helpful to visualize the collection of two sets S_A and S_B as their *characteristic matrix.*

Element	S_A	S_B
1	1	1
2	0	1
3	1	0
4	1	0
5	0	1

Figure 4 A matrix representing the sets S_A and S_B.

▶ **Example 2.** In Fig. 4 is an example of a matrix representing sets S_A and S_B chosen from the universal set $U = \{1, 2, 3, 4, 5\}$. Here, $S_A = \{1, 3, 4\}$, and $S_B = \{1, 2, 5\}$. The columns of the characteristic matrix correspond to the sets, and the rows correspond to elements of the universal set U from which elements of the sets are drawn. There is a 1 in row r and column c if the element for row r is a member of the set for column c. Otherwise the value in position (r, c) is 0. The top row and leftmost columns are not part of the matrix.

▶ **Definition 3.** Let π be a randomly chosen permutation over $[n] = \{1, 2, \dots, n\}$. For a subset $S_A \subseteq [n]$ the *min-hash* of S_A for the given permutation π, i.e., $(h_\pi(S_A))$, is a mapping of the set S_A to the element $a \in S_A$ with $\pi(a) = min\{\pi(a')|a' \in S_A\}$

In other words, the min-hash of any subset is the is the number of the first row, in the permuted order, in which the column has a 1.

π_1	S_A	S_B
1	1	1
4	1	0
5	0	1
2	0	1
3	1	0

π_2	S_A	S_B
5	0	1
4	1	0
3	1	0
2	0	1
1	1	1

π_3	S_A	S_B
3	1	0
4	1	0
5	0	1
1	1	1
2	0	1

Figure 5 The matrices representing the sets S_A and S_B after the permutations π_1, π_2, and π_3 given in the corresponding order.

▶ **Example 4.** Consider the following 3 permutations for a universe of size $n = 5$, $U = \{1,2,3,4,5\}$: $k = 1$, $\pi_1 = (1\ 2\ 3\ 4\ 5)$; $k = 2$, $\pi_2 = (5\ 4\ 3\ 2\ 1)$; $k = 3$, $\pi_3 = (3\ 4\ 5\ 1\ 2)$, where k represents the index of the permutation. Although it is not physically possible to permute very large characteristic matrices, the min-hash function h implicitly reorders the rows of the matrix of Fig. 4 so it becomes one of the other matrices given in Fig. 5.

Given the sets $S_A = \{1, 3, 4\}$ and $S_B = \{1, 2, 5\}$ the min-hashes for each permutation are as follows: $k = 1$: $h_{\pi_1}(S_A) = 1$, $h_{\pi_1}(S_B) = 1$; $k = 2$: $h_{\pi_2}(S_A) = 4$, $h_{\pi_2}(S_B) = 5$; $k = 3$: $h_{\pi_3}(S_A) = 3$, $h_{\pi_3}(S_B) = 5$. The expectation of the fraction of permutations for which the min-hashes agree is an estimation of the Jaccard similarity between the sets S_A and S_B. In this case the fraction equals to $1/3 = 0.33$ which is not a very good estimation of the true value $1/5 = 0.2$. The quality of the estimate depends on the amount of memory we are willing to use, i.e., the size of the signature, that is, the value of k.

The wonderful and simple to prove property of min-hash functions is given with the following proposition:

▶ Proposition 1. *For any pair of subsets $S_A, S_B \subseteq [n]$*

$$Pr[h_\pi(S_A) = h_\pi(S_B)] = \frac{|S_A \cap S_B|}{|S_A \cup S_B|} \pm \epsilon,$$

where the probability is defined over the random choice of the permutation π. The proof is given in [13, 14].

The on-line algorithm for estimating the similarity works in the following way: choose at random k permutations corresponding to k min-hash functions $h_1, h_2, \dots, h_k$. Then at any time t maintain: $h_i^*(S_A) = min_{j \leq t} h_i(a_j)$, and $h_i^*(S_B) = min_{j \leq t} h_i(b_j)$ for $i = 1, \dots, k$. This is simple to do in a streaming fashion: as each new element a_{t+1} from the first stream appears, the algorithm computes the min-hash $h_i(a_{t+1})$ for $i = 1, \dots, k$, i.e., for all the initially chosen permutations $\pi_1, \pi_2, \ldots, \pi_k$. Then, it compares the computed min-hashes with their current minimum $h_i^*(S_A)$. The minimum is updated only if $h_i(a_{t+1}) < h_i^*(S_A)$.

In order to maintain the $h_i^*(S_B)$ values for the second stream, the same procedure is performed simultaneously in a similar manner using the same randomly chosen k permutations. The fraction of the min-hash values that they agree on, i.e., $\hat{\sigma}(S_A, S_B) = |\{i : h_i^*(S_A) = h_i^*(S_B)\}|/k$ can be easily computed at any time.

To obtain an unbiased estimate we need to repeat the calculation multiple times, each time choosing at random k permutations. In [33] it is shown that, for $k = O(\varepsilon^{-1}\log(1/\delta))$ the value $\hat{\sigma}(S_A, S_B)$ approximates the true Jaccard similarity with probability of at least $(1-\delta)$ within a multiplicative factor of $(1 \pm \varepsilon)$. More precisely, for $0 < \varepsilon < 1$, $0 < \delta < 1$ and $k \geq 2\varepsilon^{-3}\log\delta^{-1}$, with probability of at least $1 - \delta$ holds:

$$\hat{\sigma}(S_A, S_B) \in (1 \pm \varepsilon)\frac{|S_A \cap S_B|}{|S_A \cup S_B|}.$$

The ideal family of min-hash functions is defined by the set of all permutations over $[n]$. Storing a single permutation from this family requires $\mathcal{O}(n\log n)$ bits; hence, they are not suitable for data stream applications. In [56] a family of *approximate* min-hash functions is presented such that any function from this family can be represented using $\mathcal{O}(\log n \log(1/\epsilon'))$ bits (each hash function being computed efficiently in $\mathcal{O}(\log(1/\epsilon'))$ time). This induces an additional error to the approximation of ϵ', for which only the k value will have to be appropriately adjusted. The advantage is in the savings of space and time.

5.2 Estimating Similarity on Windowed Data Streams

In many real-life scenarios the users are most interested on the most recent statistics or models gathered over the "recently observed" data elements. Despite the exponential growth in the storage capacity of the available systems, it is not common for such streams to be stored even partially. For example, consider high-speed, backbone Internet routers that route several Gbit/s and process tens of millions of packets per second on average. Storing the log of these packets locally even for an hour will require several MB of fast memory. Alternatively, moving it to a central warehouse would consume a sizable portion of the network bandwidth [28]. The windowed data stream model was formalized as a framework for designing algorithms, addressing the need of reasoning in this context.

Let us briefly recall the definition of the *combinatorial* [58] or *sliding window* data stream model defined previously in Section 3.3: At any time t consider the window of the last w observations $a_{t-(w-1)}, a_{t-(w-2)}, \ldots, a_t$, where each item a_i is a member of the universe of n items U. In this model we are allowed to ask queries about the data in the window using only $o(w)$ (often polylogarithmic in w) storage space.

Using sliding windows causes additional complications since maintaining simple statistics like minimum or maximum over a window of most recent data requires storing the most recent t items in the window, i.e., when a new item comes in, an old item is removed. Despite these difficulties, there are algorithms for estimating both the L_p distance [29] and the Jaccard similarity [28] over sliding windows of streaming data.

5.2.1 Approximating the L_p Norm

The work of Datar et al. [29] provides a general method for translating a wide range of data stream algorithms into the windowed data stream models, such as maintaining histograms, hash tables, distinct values and statistics or aggregates such as averages/sums.

Their technique is based on a special type of a histogram called *exponential histogram*, which is used to partition the window of w items into buckets. The main property of their algorithm is to maintain buckets with exponentially increasing sizes such that: there are at most $\frac{k}{2}+1$ and at least $\frac{k}{2}$ buckets of each bucket size, where $k = \lceil \frac{1}{\epsilon} \rceil$. Thus, whenever there are $\frac{k}{2}+2$ buckets of same size, the oldest two buckets are merged, which may occasionally lead to a cascade of such mergers.

Each bucket maintains the L_p sketches computed over the items it contains, but not the actual values of the items. Additionally, for each bucket a time-stamp is associated that marks the oldest *active* element in the bucket, and is used to indicate expiry. The expired buckets are deleted, and new ones are created for the newly encountered items. The absolute error of their estimate is due to the fact that the last bucket may contain items older than the last observation seen at time $t-(w-1)$. However, this error is bounded with an additive $(1+\epsilon)$ factor loss in accuracy for a multiplicative overhead of $\mathcal{O}(\frac{1}{\epsilon}\log w)$ in memory. The query time for the exponential histogram is $\mathcal{O}(1)$, and the worst-case processing time is $\mathcal{O}(w)$. For more details the reader is referred to [29].

5.2.2 Approximating the Jaccard Similarity

Computing an approximation of the Jaccard similarity in the windowed data stream model is not an easy task, primarily due to the problem of maintaining the minimum over a sliding window. The algorithm for approximating the Jaccard similarity described in section 5.1.2 requires at any time the correct minimal values for each hash function $h_i^*(A)$ $(h_i^*(B))$, for $i = 1, \ldots, k$ computed after every new observation a_t (b_t). To be able to maintain these minimums, one needs to store the outcome of all $h_i(a_j)$ $(h_i(b_j))$, where $j = t-(w-1), \ldots, t$. Thus, the problem boils down to maintaining the minimum hash values $h_i^*(t)$ for each random permutation at any time t for both of the streams. Datar and Muthukrishnan [28] provide a simple solution to this problem based on the idea of maintaining a linked list of hash values h_i and their timestamps only for the *dominant* items. Note, that we are interested only in the items which are appearing in the current window, i.e., with timestamps greater than or equal to $t-(w-1)$.

The property of *dominance* is defined as follows: Consider at time t two items d_1 and d_2 from the window of most recent w items with arrival times t_1, t_2 such that $t_1 < t_2 < t$. If $h_i(d_1) \geq h_i(d_2)$ then we say that item d_2 dominates item d_1. Thus, as long as there is an item d_2 that dominates an item d_1 both appearing in the window, we need not to store the hash value $h_i(d_1)$. It is easy to see that, at any time t if d_1 is in the window then d_2 is also present and has a hash value no greater than $h_i(d_1)$. Hence, the minimum h_i^* at time t will not be affected by the hash value of an item that is dominated.

Based on the observation above, the authors propose to maintain at any time t a linked list $L_i(t)$ for all $i = 1, \ldots, k$ permutations. Every element of this list will represent a pair of a hash value and its time-stamps $(h_i(a_j), j)$ for some data item a_j at time t, where j is the arrival time of the item and satisfies the property $t-(w-1) \leq j \leq t$. The list would look like:

$$\{(h_i(a_{j_1}), j_1), (h_i(a_{j_2}), j_2), \ldots, (h_i(a_{j_l}), j_l)\}, \text{ where } l \leq w.$$

The list satisfies the property that both the hash values and the arrival times are strictly increasing from left to right, i.e.,

$$j_1 < j_2 < j_3 < \ldots < j_l \text{ and } h_i(a_{j_1}) < h_i(a_{j_2}) < h_i(a_{j_3}) < \ldots < h_i(a_{j_l}),$$

where clearly $h_i^*(t) = h_i(a_{j_1})$.

Maintaining this list is simple. When a new data item arrives a_{t+1}, we compute its hash value $h_i(a_{t+1})$ and traverse the list $L_i(t)$ looking for the largest index j' (eg., a binary search over a special data structure) such that $h_i(a_{j'}) \leq h_i(a_{t+1})$. Then, we remove all the pairs from the list which appear after the pair with index j'. Now, $(h_i(a_{j'}),j')$ will be the rightmost item in the list $L_i(t)$:

$$j_1 < j_2 < \ldots < j' \text{ and } h_i(a_{j_1}) < h_i(a_{j_2}) < \ldots < h_i(a_{j'}),$$

If its hash value is different from $h_i(a_{t+1})$ then we insert the pair $(h_i(a_{t+1}),\ t+1)$ at the end of the list $L_i(t)$. Otherwise, we only need to update $(h_i(a_{j'}),j')$ into $(h_i(a_{j'}),t+1)$. The last step is to check if the leftmost pair corresponds to an item which is still present in the window. If $j_1 \notin \{(t+1)-(w-1),\ldots,t+1\}$ than the leftmost pair is deleted, and we get an updated list $L_i(t+1)$.

In the worst case this procedure will require a memory of $\mathcal{O}(w)$. However, with high probability, over the random choice of min-hash functions h_i the size of the list is proportional to the Harmonic number H_w, given by $1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{w} = \Theta(\log w)$. A proof of this bound is given in Section 3.3. Hence, the standard algorithm can be adapted to the windowed data stream model, using $\mathcal{O}(\log w + \log n)$ words of space and taking $\mathcal{O}(\log\log w)$ processing time per data item, with high probability. Note that, the success of the result above is predicated on the random choice of the random min-hash functions, and not over the distribution of the input, meaning that, it holds for an arbitrary (worst case) input.

6 Group Testing for Tracking Frequent Items

Tracking the *hot* items (those that occur frequently) on an underlying database relation or a data stream is a fundamental issue for the task of continuous query selectivity estimation, *iceberg* query computation or simple outliers detection in stream data mining. Hot items influence caching, load balancing, network traffic management, market-basket analysis and are crucial for a successful anomaly detection. As such, maintaining the set of hot items at any time is an interesting and an important problem. Formally, the question is how to dynamically maintain a set of hot items under the presence of delete and insert operations in the general *turnstile* data stream model.

Imagine that you observe a sequence (or a stream) of m operations on items, each member of a universe U of size n. Without loss of generality, we can assume that the item identifiers are integers in the range 1 to n. The net occurrence of any item x at time t, denoted $c_x(t)$, is the number of times the item x has been inserted minus the number of times it has been deleted. The current frequency of any item is thus given by:

$$f_x(t) = c_x(t) / \sum_{i=1}^{n} c_i(t).$$

The k most frequent items at time t are those with the k largest $f_x(t)$'s, where k is a parameter. On the other hand, an item x is called *hot item* if $f_x(t) > 1/(k+1)$, i.e., it represents a significant fraction of the entire dataset. Clearly, there can be at most k hot items, and there may be *none*.

▶ **Example 5.** Observe the following sequence of items: 1,2,1,3,4,5,1,2,2,3,1,1,3,5,2,6,1,2 each of them being a member of the set [1, 6]. Their corresponding frequencies are: $f_1 = 6/18$, $f_2 = 5/18$, $f_3 = 3/18$, $f_4 = 1/18$, $f_5 = 2/18$ and $f_6 = 1/18$. For $k = 3$, hot items are only 1 and 2 ($f_1 = 6/18 = 1/3 > 1/4$ and $f_2 = 5/18 > 1/4$).

6.1 Preliminaries

Determining the set of hot items is an easy problem if we are allowed a memory of $\mathcal{O}(n)$ words. Using a simple heap structure, we can process each *insert* or *delete* operation in $\mathcal{O}(\log n)$ time and find the hot items in $\mathcal{O}(k \log n)$ time in the worst case [4]. As discussed in several occasions for many streaming applications it is important to use sub-linear space $o(n)$ on the cardinality of the data stream. However, Alon et al. [6] proved that estimating $f^*(t) = max_x f_x(t)$ is impossible with $o(n)$ space. Thus, estimating the k most frequent items or the k hot items is at least as hard. A simple argument from information theory can help us show that solving this problem *exactly*, i.e., finding *all* and *only* items which have frequency greater than $1/(k+1)$ requires the storage of at least n bits.

This also applies to randomized algorithms. Any algorithm which guarantees to output all hot items with probability at least $1-\delta$, for some constant parameter δ, must also use $\Omega(n)$ space. This follows by observing that the above statement corresponds to the Index problem in communication complexity [50]. However, if we are willing to accept approximate answers it is possible to guarantee with high success probability at least $1-\delta$, that *all* hot items will be found and no item which has frequency less than approximately $\frac{1}{k+1} - \epsilon$, for some user-specified parameter ϵ and any user-specified probability δ [25].

6.2 Background

The problem of finding the most frequent items in one-pass with *limited* storage has gained a lot of interest in the last two decades. There is a large body of one-pass algorithms for finding the k most frequent items in the simpler data stream model in which only insert operations are allowed [30, 48, 52]. The general idea is to hold a number of counters (polylogarithmic in n), each associated with a single item seen in the sequence. The counters are incremented whenever their corresponding item is observed, but are decremented or deallocated only under certain circumstances. Therefore, they cannot be easily adapted to the dynamic case. As before, the algorithms guarantee that all hot items will be found, including items about which *no guarantees* of frequency can be made.

Another approach is to use filters: as each item arrives, the filter is updated to account for this arrival. Items which are above the threshold are retained as possible candidates for hot items. At output time all the retained items are rechecked with the filter, and those that pass the filter are output. Filter methods can only discover items when they become *hot* but cannot retrieve items from past which have since become *frequent* [25]. An important result that is of Charikar et al. [20], who gave an algorithm to approximate the count of any item correct up to ϵn in $\mathcal{O}(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ space and $\mathcal{O}(\log \frac{1}{\delta})$ time per update.

In the general turnstile model only the algorithms proposed by Cormode and Muthukrishnan [25] give theoretical space and time guarantees, which are outperformed in practice, as claimed. The algorithms use $\mathcal{O}(k \log k \log n)$ space for a summary data structure, and are able to process each transaction in $\mathcal{O}(\log k \log n)$ time. Querying the summary for finding the hot items takes $\mathcal{O}(k \log k \log n)$ time, which is independent on the size of the stream.

The approach of Cormode and Muthukrishnan is based on two different "group testing" procedures which can be categorized as *adaptive* and *nonadaptive.* Thus the methods are different in nature and give slightly different time and space guarantees. In the following subsection we will discuss the nonadaptive group testing method which is more efficient for the case of high transaction rates. For the adaptive group testing based method the interested reader is referred to [25].

6.3 Nonadaptive Group Testing

The general idea behind the algorithm is to randomly create $\mathcal{O}(k \log k)$ groups or sets of items, which are further deterministically divided or grouped into $\mathcal{O}(\log n)$ subgroups using error-correcting codes. Each group is associated with a counter which is incremented whenever an item that belongs to that group is inserted, or decremented when the item gets deleted. If a group contains a hot item then its corresponding counter will exceed a certain threshold. Thus, discovering the hot items requires an assembling of all the results from the tests performed over the different groups.

To ease the exposition of the algorithm, we will first describe a solution to the simpler problem of finding the *majority* (occurs more than half of the time) item. The algorithm for solving the latter problem will then be extended for the problem of finding k hot items.

6.3.1 Finding the Majority Item

While finding the majority item in the *cash-register* data stream model (where only insert operations are allowed) is easy, this problem looks less trivial for the *turnstile* data stream model. The reason is that an item which was found to be frequent, can become infrequent due to a sequence of delete operations. However, there exist a deterministic algorithm to solve this problem using $\lceil \log_2 n \rceil + 1$ counters. The algorithm maintains the set of counters at any time trough increment and decrement operations. To identify the majority item at output time a binary search procedure of $\log(n)$ steps is used.

The first counter $d_0 = c(t) = \sum_x c_x(t)$ keeps track of the number of items in total, i.e., we increment d_0 for every insert and decrement it for every delete operation. The remaining counters d_1, d_2, …, d_j are associated each with its corresponding jth bit of the binary representation for a given item identifier x in the range 1 to n. Thus, j goes from 1 to $\log n$. Let $bit(x, j)$ is a function that returns the value of the jth bit of the binary representation of the integer x. The update procedure is as follows:

On *insert(x)*: increment d_0 and update all counters c_j with $+bit(x, j)$, for $j = 1, \ldots, \log n$
On *delete(x)*: decrement d_0 and update all counters c_j with $-bit(x, j)$, for $j = 1, \ldots, \log n$

At output time the algorithm does a binary search over the set of counters. The logic is simple: if there is an item whose count is greater than $d_0/2$ (majority item), then for any way of dividing the elements into two sets, the set containing the majority item will have weight greater than $d_0/2$, and the other will have weight less than $d_0/2$. For example, if $c_1 > c_0/2$ (the least significant bit) means that the majority item is an item from the group of odd numbers. Next we will need to examine if it belongs to the group of items divisible with 4 or not, i.e., we need to test the value of c_2. In this way we proceed with examining the values of all the counters using the following procedure:

```
Initialize $x \leftarrow 0$
For $j = 1, \ldots, \log n$, if $c_j > c_0/2$ then $x \leftarrow x + 2^{j-1}$
Output $x$
```

The algorithm described above guarantees always to find the majority item if there is one. If there is none such item, it will still return some item. Note, that in that case it will not be possible to distinguish the difference based only on the information stored.

6.3.2 Finding k Hot Items

Suppose we have selected a group of items to monitor which happened to contain only one hot item. Then we can apply the algorithm from the previous section to this group by dividing it further into $\log n$ buckets and associating a counter with each bucket. Determining the hot item in the group would require simple "weighting" of all the buckets. Provided that the total weight of other items in the group is not *too much* the hot item will always be in the heavier of the two buckets.

The idea is to divide each group into $\log n$ subgroups in which we will not hold an exact count for each separate item that is mapped to the group. This enables us to use space polylogarithmic in the size of the universe, albeit on the cost of an approximate answer. However, as we shall see this approximation can be very close to the correct answer. The choice of items belonging to each group can be done completely randomly, in which case we would have to store a list of members for every group explicitly (space at least linear in n). Instead, to create a concise description of each group, one may use hash functions which will provide a mapping of items to the groups. Each group will consist of all the items which are mapped to the same value by a particular hash function. The advantage of this approach comes from the possibility to store a concise representation for each hash function, using space $\mathcal{O}(\log n)$.

Let assume that we need W groups each divided further into $\log n$ subgroups. We need a hash function which will provide a mapping from the set of item identifiers $[1, n]$ to the set of group identifiers $[1, W]$. The hash functions used in the algorithm of Cormode and Muthukrishnan [25] are universal hash functions derived from those given by Carter and Wegman [16], and were discussed in Section 4.2. Briefly, each hash function is defined by a and b, which are integers smaller than P (initially chosen to be $\mathcal{O}(n)$) as: $f_{a,b}(x) = (((ax+b) \bmod P) \bmod W)$, where $P > n > W$ is a fixed prime, and a and b are drawn uniformly at random in the range $[0, P-1]$. As a result, the space required to store each hash-function representation is $\mathcal{O}(\log n)$ bits.

When a hash function is used to provide a mapping into a number of groups, there is a probability that two different items will be mapped to the same group. The authors make use of the following fact which comes from *Proposition 7* of [16]:

Over all choices of a and b, for $x \neq y$, $\Pr[f_{a,b}(x) = f_{a,b}(y)] \leq \frac{1}{W}$.

This probability is directly connected with the success probability of the algorithm for determining all k hot items. Therefore, we need to maximize it by using not one but several hash functions of this type. Lets assume that we will use T such hash functions. As a result we get $T \times W$ overlapping groups. For storing the representation of each hash function h_i we will need two arrays $a[1 \ldots T]$ and $b[1 \ldots T]$ whose values are chosen at random, having $h_i = f_{a[i],b[i]}$ for $i = 1, \ldots T$.

The data structure which will be maintained at all times is a three-dimensional array of counters d, of size $T \times W \times (\log n + 1)$. In addition to that, we need a counter for the current total number of items seen m. The counters $d[1][0][0]$ to $d[T][W-1][\log n]$ are all initialized to zero. The counter $d[0][0][0]$ is used to keep count of the total number of items. Let $G_{i,j} = \{x | h_i(x) = j\}$ be the set of item identifiers which will be mapped to group $G_{i,j}$ by the hash function h_i, for $i = 1, \ldots T$ and $j = 1, \ldots W$. To keep the count of the current number of items within each group $G_{i,j}$ we will use the counters $d[i][j][0]$. For each such group we will need $\log n$ counters for $\log n$ subgroups defined as $G_{i,j,l} = \{x | x \in G_{i,j} \wedge bit(x, l) = 1\}$. These correspond to the groups used for finding the majority item. We will use $d[i][j][l]$ to keep count of the current number of items within subgroup $G_{i,j,l}$.

The update procedure is simple and very similar to the update procedure for finding the majority item, i.e., update the $\log n$ counters for each of the groups where an item x belongs to based on its bit representation in exactly the same way as in section 6.3.1. The time to perform an update $\mathcal{O}(T \log n)$ is the time taken to compute the T hash functions, and to modify $\log n$ counters for each of those T mappings. To output the hot items the structure can be searched at any time. The basic test wold be whether the count for a group or a subgroup exceeds the threshold needed for an item to be hot, which is $m/(k+1)$. A group containing a hot item will always pass this test, but the same is possible for a group which does not contain a hot item. Although this probability is very small various checks need to be made in order to reduce the number of items output which are not hot.

The search procedure consist of examining all of the groups and testing if they contain a hot item. That is, for a given group $G_{i,j}$, if $d[i][j][0] \leq m/(k+1)$ then there cannot be a hot item in that group, and the group is rejected. For the groups which are not rejected we need to examine the counts of their subgroups. If a group is not rejected, then there is enough information to discover the identity of the hot item x contained. At the end, the discovered hot item need to be further verified if it belongs to the group it was found in, and if all the groups where the item belongs are above the threshold, i.e., $d[i][h_i(x)][0] > m/(k+1)$ for all i. The total time to find all hot items is $\mathcal{O}(T^2 W \log n)$.

Cormode and Muthukrishnan [25] gave the following final result: *Choosing* $W \geq 2/\epsilon$ *and* $T = \log_2(k/\delta)$ *for a user-specified parameter* δ *ensures that, with probability at least* $1 - \delta$ *we can find all hot items whose frequency is more than* $\frac{1}{k+1}$*, and for a given* $\epsilon \leq \frac{1}{k+1}$*, with probability at least* $1 - \delta/k$ *each item which is output has frequency at least* $\frac{1}{k+1} - \epsilon$*.*

The proof is simple and is based on the property of the hash functions used and the Markov inequality, combined with some simple observations on the testing procedure. The interested reader is referred to [25] for more details. If we substitute the values for T and W from the above result into the previously given time and space bounds we will obtain the following bounds: the upper bound on the space required is $\mathcal{O}(\frac{1}{\epsilon} \log n \log(k/\delta))$, the update time takes $\mathcal{O}(\log n \log(k/\delta))$, and the query time is no more than $\mathcal{O}(\frac{1}{\epsilon} \log 2(k/\delta) \log n)$.

One of the drawbacks of the method is that the update time depends on the product of T and $\log n$, which can be slow for streams with high cardinality, i.e., large item identifiers. To reduce the time dependency on T each of the hash functions can be applied in parallel, and the relevant counts can be modified separately. The dependency on $\log n$ can be addressed by increasing the space usage. The observation is that, if instead of using the function $bit(x,i)$ one can use a function $dig(x,i,b)$ which gives the ith digit in the integer x when it is written in base $b \geq 2$. Then, within each group one will need to keep $(b-1)$ $\times \log_b n$ subgroups: the i,j group now counting how many items have $dig(x,i,b) = j$ for $i = 1, \ldots, \log_b n$ and $j = 1, \ldots, b-1$. Setting b to m will correspond to keeping a count for every item.

7 Clustering and Summarizing Data Streams

In this last section we will discuss some more advanced algorithms for solving basic summarization problems in the singe-pass data stream scenario. We will place the focus on the *maximum error* histogram construction problem, although the techniques discussed here can be applied to other summarization problems like K-center, K-median clustering and VOPT histogram construction [39].

Histograms and related synopsis structures are popular techniques for approximating data distributions, and may serve as basic building blocks in the design of more sophisticated summary data structures for maintaining other statistics of interest. They have been extensively used in database query optimization for estimating selectivity factors [57] and access path selection in a relational database management system [61], in approximate query answering [1], mining time series [18], and many other areas.

With the increased interest into mining data streams several streaming algorithms for histogram construction problems have been proposed [38, 39, 41, 15, 49]. However, all of them have space bounds dependent on the size of the input m, the magnitude of the optimum solution ε^* or the machine precision M. A new result given by Guha [37] improves all previous algorithms on the problems of histogram construction and K-center clustering in either the space bound, the approximation factor or the running time. It represents the best algorithm applicable to streaming scenarios with *tunable* guarantees, linear running time and memory requirements independent of the input size.

In the following sections we will describe three main ideas used in the framework proposed by Guha [37]: (1) the notion of "thresholded approximation", (2) the idea of running multiple copies of the algorithm corresponding to different estimates of the final error and, (3) "streamstrapping" as a way to bootstrap the estimation procedure by using the summaries of the prefixes of the data to choose the correct granularity required to further inspect the data.

7.1 Maximum Error Histograms

Let $X = x_1, \ldots, x_m$ be a finite data sequence of m real-valued numbers. The histogram construction problem is defined as follows: given some space constraint B, create and store a piecewise constant representation H_B of the data sequence using at most B storage (pieces), such that H_B is optimal under some notion of error $E_X(H_B)$ defined between the data sequence and H_B. The representation is a grouping of the values of consecutive points x_i, where $i \in [l_j, r_j]$ into a single value h_j, thus forming a bucket b_j, defined with the smallest l_j and the greatest r_j index of the data points belonging to the bucket, and their representative value h_j. In other words, for $l_j \le i \le r_j$ we estimate x_i by h_j. The histogram uses at most B buckets which cover the entire interval $[1, m]$, and saves space by storing only $\mathcal{O}(B)$ numbers instead of m.

Since h_j is an estimate for the values in bucket b_j, for the query at point i, where $l_j \le i \le r_j$ we incur an error $x_i - h_j$. The error $E_X(H_B)$ of the histogram H_B is defined as a function of these point errors. Since the interval corresponding to the buckets do not overlap and every point belongs to exactly one bucket, we can express the total error of a histogram H_B with buckets $b_1, \ldots, b_B$ as a sum over all bucket errors: $\sum_j Err(b_j)$. In the case of the maximum (absolute) error histogram construction problem, the error Err for the bucket b_j defined by the interval $[l_j, r_j]$ and representative h_j is defined as follows:

$$Err(b_j) = Err(l_j, r_j) = \max_{i \in [l_j, r_j]} |x_i - h_j|.$$

The error of the histogram is given with:

$$E_X(H_B) = \sum_j Err(b_j) = \sum_j \max_{i \in [l_j, r_j]} |x_i - h_j|.$$

In order to minimize the error of the histogram we need to minimize the error of each bucket. For the case of the maximum **absolute** error histogram construction problem the representative values which minimize the maximum absolute error are computed using the formula $h_j = \frac{x_{l_j} + x_{r_j}}{2}$. Replacing the formula for h_j in $x_i - h_j$ the error of a bucket is given with $Err(b_j) = \frac{x_{r_j} - x_{l_j}}{2}$.

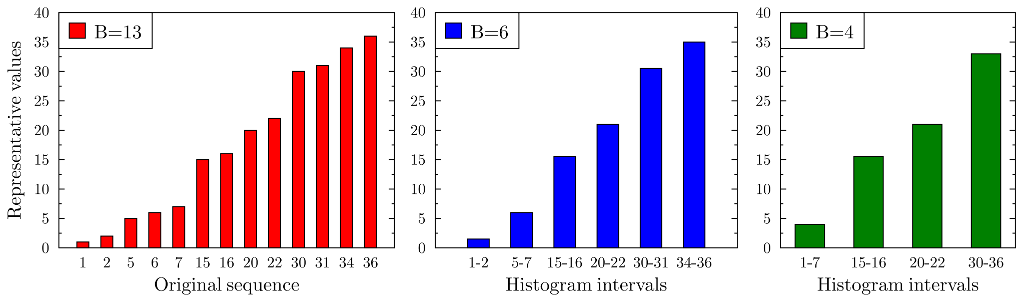

Figure 6 Illustration of the histogram construction problem.

► **Example 6.** Let us consider the following data sequence $X = \{20, 1, 5, 15, 5, 2, 16, 22, 36, 30, 34, 7, 31\}$. When constructing histograms it is usually useful to order the data $X = \{1, 2, 5, 6, 7, 15, 16, 20, 22, 30, 31, 34, 36\}$ to ease the representation. Figure 6 illustrates the original sequence and the resulting histograms for the given sequence X and two possible storage constraints: $B = 6$ and $B = 4$. As we can see, each histogram tries to approximate the data sequence using fewer data points. Thus, the histogram construction algorithm has to examine all of the possible divisions of this sequence in order to find the one that minimizes the total error $E_X(H_B)$ of the histogram.

We would first like to construct a maximum absolute error histogram using at most B=6 storage, which means that our histogram will have not more than 6 buckets. Let assume that we have such an algorithm which is able to find the optimal histogram for the given sequence X and space constraints B. The resulting optimal histogram for B=6 divides the data elements in the following sequence of buckets $B_1 = \{1, 2\}$, $B_2 = \{5, 6, 7\}$, $B_3 = \{15, 16\}$, $B_4 = \{20, 22\}$, $B_5 = \{30, 31\}$ and $B_6 = \{34, 36\}$, represented with the corresponding sequence of values $h_1 = 1.5$, $h_2 = 6$, $h_3 = 15.5$, $h_4 = 21$, $h_5 = 30.5$ and $h_6 = 35$, with a cumulative error $E_X(H_B) = 4$.

However, if we wish to reduce the storage to $B = 4$ pieces, then there are two possible solutions with equal errors and one of them comprises the following sequence of buckets $B_1 = \{1, 2, 5, 6, 7\}$, $B_2 = \{15, 16\}$, $B_3 = \{20, 22\}$ and $B_4 = \{30, 31, 34, 36\}$, represented with the corresponding sequence of values $h_1 = 4$, $h_2 = 15.5$, $h_3 = 21$ and $h_4 = 33$. The total error under these space constraints is $E_X(H_B) = 8$.

Another variant of the maximum error histogram construction problem is to use the relative error instead:

$$Err(l_j, r_j) = \min_{h_j} \max_{i \in [l_j, r_j]} \frac{|x_i - h_j|}{\max\{c, |x_i|\}},$$

where c is an absolute constant that works as a sanity bound, used to reduce excessive domination of the relative error by small data values. By setting c to be larger than all numbers in the input, the relative error is reduced to an absolute error multiplied by $\frac{1}{c}$ which allows to discuss both errors at the same time. The maximum absolute or relative error metrics enable to approximate the data with uniform fidelity throughout the domain, unlike sum-based measures.

Jagadish et al. [46] first gave a general technique for computing the optimum histogram in $\mathcal{O}(m^2 B)$ time and $\mathcal{O}(mB)$ space for several measures. However, the quadratic running time showed is undesirable for large data sets, not to mention for streaming applications. Besides, having in mind that the histogram is already an approximation of the data, it came natural to think of near optimal solutions which can be constructed in time linear in the size of the input data. This line of thinking went even further, considering "tunable" approximations which would allow faster running times if a less accurate histogram suffices for the application at hand. As a result, many solutions have been developed for a slightly different problem formulation known as constructing $(1+\epsilon)$-*approximate* histograms [39]: *Given a sequence X of length m, a number of buckets B, and a precision parameter $\epsilon > 0$, find H_B with $E_X(H_B)$ at most $(1+\epsilon)\min_H E_X(H)$ where the minimization is taken over all histograms H with B buckets.* Thus, if we desire a 1% approximation to the optimal histogram, we would set $\epsilon = 0.01$. However, all of the solutions proposed have running times dependent on the size of the input, although polylogarithmic.

An interesting and important result is given by Guha and Shim [40], where they present a linear time *optimal* algorithm for the maximum absolute and relative error measures for computing the optimum histogram in $\mathcal{O}(m + B^2 \log^3 m)$ time and $\mathcal{O}(m)$ space. Note that the improvement in the running time is on the cost of an increased memory usage. Although the algorithm is linear, due to its memory dependence on the size of the input it cannot be used in streaming applications.

Only recently Guha [37] gave the first tight results for linear time algorithms whose space requirements do not depend on the size of the input, i.e., the data stream. The *StreamStrap* algorithm [37] uses the concept of thresholded approximation plugged into a framework in which multiple repetitions of the thresholded algorithm are run in a sequential manner, where each new run is enhanced and bootstrapped with the results from the previous run, an idea called "streamstrapping". The StreamStrap algorithm is discussed in more detail in the following section.

7.2 The StreamStrap Algorithm

To be able to apply the StreamStrap algorithm there are two basic requirements which have to be fulfilled in our summarization scenario:

1) *Thresholded small space approximations exists.*
2) *The error measure is a Metric error.*

For a given summarization problem P a *thresholded approximation* (as given in the work of [37]) is defined to be an algorithm which simultaneously guarantees that: 1) if there is a solution with summarization size B' and error ε (where ε is known), then in small space we can construct a summary of size at most B' such that the error of our summary is at most $\alpha\varepsilon$ for some $\alpha \geq 1$ and, 2) otherwise declare that no solution with error ε exists. An important point of the first requirement is that we use the knowledge of ε.

The second requirement translates into a property of the error measure which enables us to use the following inequality for any X, Y, H, H':

$$Err(X(H) \circ Y, H') - Err(X, H) \leq Err(X \circ Y, H') \leq Err(X(H) \circ Y, H') + Err(X, H),$$

where $Err(X, H)$ is the summarization error of X using the summary H (eg., maximum error histogram), $X \circ Y$ denotes a concatenation of input X followed by Y, and $X(H)$ is the summarized input in which every point x is replaced by the corresponding representative h from H.

A *thresholded* version of the optimal algorithm for the maximum error problem [40] can be easily derived by using the knowledge of ε in the computations [37]. For a given error ε the algorithm can produce a summary of the input in linear time, using $\mathcal{O}(B')$ space, if such summary exists. Further, both the maximum error and the square root of the VOPT error satisfy the second requirement. Thus, we can fulfill all the conditions required for the StreamStrap algorithm to be applied.

On a high level the algorithm proceeds as follows: First, it reads B points from the input. Since all the input values are stored at this point of time, the summarization error is 0. The algorithms continues with reading as long as the error remains zero. When the first input which causes a non-zero error is observed (say this error is ε_0), the algorithm initializes J copies of the *thresholded* summarization algorithm and runs the algorithms. Each copy is run for a different error value which is exponentially increasing with a factor of $(1+\epsilon)$, i.e., ε_0, $(1+\epsilon)\varepsilon_0$, …, $(1+\epsilon)^J\varepsilon_0$. The value of J is chosen such that $(1+\epsilon)^J > \alpha/\epsilon$, giving us $\mathcal{O}(\frac{1}{\epsilon}\log\frac{\alpha}{\epsilon})$ different algorithms.

A property of the thresholded algorithm is that it will succeed only if a summary exists for the given error ε and the available storage space. Therefore, when at some point in time some of the copies of the thresholded algorithm will declare a "fail" for some ε', we know that there exist no such solution for an error smaller or equal to ε', that is, $\varepsilon^* > \varepsilon'$. Now, we terminate all the algorithms run for error estimates $\leq \varepsilon'$ and, start running a thresholded algorithm for $(1+\epsilon)^J\varepsilon'$ using the summary from the "failing" algorithm as the initial input. Whenever a running copy of the thresholded algorithm will declare a "fail" the same procedure is applied. As a result, we will always have the same number of running algorithms, but for different error estimates. At query time the algorithm returns the answer for the lowest error estimate for which a thresholded algorithm is still running, i.e., have not declared a "fail".

The general idea is to start with the smallest possible estimate of the error and raise it a number of times until we find a solution under the constraints on the memory storage and the approximation factor ϵ. Using the summaries from the previous runs and the property of a Metric error, it can be shown that for a given summarization problem that fulfills the requirements of the framework, for any $\epsilon \leq 1/10$ the StreamStrap algorithm provides a $\alpha/(1-3\epsilon)^2$ approximation. The proof is given in [37].

7.3 Applications

If we apply the StreamStrap algorithm for the maximum error histogram construction problem with B buckets, we can have a single pass $1+\epsilon$ *streaming* approximation using $\mathcal{O}(\frac{B}{\epsilon}\log\frac{1}{\epsilon})$ space and $\mathcal{O}(m + \frac{B}{\epsilon}(\log^2\frac{B}{\epsilon})\log M\varepsilon^*)$ time. The error of any bucket will be additively within $\epsilon\varepsilon^*$ of the true error of that bucket.

The StreamStrap algorithm has been applied also to the the problem of K-center clustering, K-median clustering and the VOPT histogram construction problem, for which upper bounds on the space and running time are given [37]. Guha further proved the first lower space bounds for maximum error histograms and for the K-center problem in the *Oracle Distance Model*, where an oracle is assumed which given two input points and an additional small space determines their distance. In particular, for the maximum error histogram construction problem he proved that: *for all $\epsilon \leq 1/(40B)$, any $1+\epsilon$ approximation for B bucket maximum error histogram, which also approximates the error of each bucket within additive ϵ times the optimum error must use $\Omega(\frac{B}{\epsilon\log(B/\epsilon)})$ bits of space.* This result is proved by using a reduction of the Indexing problem to the problem of constructing a histogram.

The importance of Guha's results lies in the fact that the StreamStrap algorithm can run indefinitely using a bounded amount of space and a constant processing time for each data item from the stream, while still providing an approximation or a summary which is according to the user's specifications (given the approximation factor ϵ and the size of the summary B). Thus, it can be easily applied in many data stream applications in the *cash-register* model, and as a building block in more sophisticated summary data structures for tracking various statistics of interest. To the best of our knowledge, similar results have not yet been achieved for the more general *turnstile* data stream model.

References

1 Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. *SIGMOD Rec.*, 28:574–576, June 1999.

2 Gagan Aggarwal, Mayur Datar, Sridhar Rajagopalan, and Matthias Ruhl. On the streaming model augmented with a sorting primitive. *Foundations of Computer Science, Annual IEEE Symposium on*, pages 540–549, 2004.

3 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley Publishing Co., Reading, Mass.-London-Amsterdam, 1975.

4 Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft. *Data structures and algorithms.* Addison-Wesley, Reading, Mass., 1983.

5 Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS'99, pages 10–20, New York, NY, USA, 1999.

6 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

7 C.R. Aragon and R.G. Seidel. Randomized search trees. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:540–545, 1989.

8 Aditya Akella Ashwin, Ashwin Bharambe, Mike Reiter, and Srinivasan Seshan. Detecting ddos attacks on isp networks. In *Proceedings of the Workshop on Management and Processing of Data Streams*, 2003.

9 Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS'02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002.

10 Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA'02, pages 633–634, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

11 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702 – 732, 2004. Special Issue on FOCS 2002.

12 Vladimir Braverman, Rafail Ostrovsky, and Carlo Zaniolo. Optimal sampling from sliding windows. *J. Comput. Syst. Sci.*, 78(1):260–272, 2012.

13 A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES'97, pages 21–29, Washington, DC, USA, 1997. IEEE Computer Society.

14 Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proceedings of 13th Annual ACM Symposium on Theory of Computing*, STOC'98, pages 327–336, New York, NY, USA, 1998. ACM.

15 C. Buragohain, N. Shrivastava, and S. Suri. Space efficient streaming algorithms for the maximum error histogram. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1026 –1035, april 2007.

16 J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC'77, pages 106–112, New York, NY, USA, 1977. ACM.

17 CERN: European Organisation for Nuclear Research. `http://public.web.cern.ch/`.

18 Kaushik Chakrabarti, Eamonn Keogh, Sharad Mehrotra, and Michael Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Trans. Database Syst.*, 27:188–228, June 2002.

19 Timothy M. Chan and Eric Y. Chen. Multi-pass geometric algorithms. *Discrete & Computational Geometry*, 37:79–102, 2007.

20 Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP'02, pages 693–703, London, UK, UK, 2002. Springer-Verlag.

21 Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55:441–453, December 1997.

22 Edith Cohen, Mayur Datar, Shinji Fujiwara, Aristides Gionis, Piotr Indyk, Rajeev Motwani, Jeffrey D. Ullman, and Cheng Yang. Finding interesting associations without support pruning. *IEEE Trans. on Knowl. and Data Eng.*, 13:64–78, January 2001.

23 Edith Cohen and Martin Strauss. Maintaining time-decaying stream aggregates. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS'03, pages 223–233, New York, NY, USA, 2003. ACM.

24 Graham Cormode, Mayur Datar, Piotr Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB'02, pages 335–345. VLDB Endowment, 2002.

25 Graham Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS'03, pages 296–306, New York, NY, USA, 2003. ACM.

26 Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58 – 75, 2005.

27 Corinna Cortes and Daryl Pregibon. Giga-mining. In *Knowledge Discovery and Data Mining*, pages 174–178, 1998.

28 M. Datar and S. Muthukrishnan. Estimating rarity and similarity over data stream windows. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA'02, pages 323–334, London, UK, UK, 2002. Springer-Verlag.

29 Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31:1794–1813, June 2002.

30 Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In Rolf H. Möhring and Rajeev Raman, editors, *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 348–360. Springer, 2002.

31 Earth System Research Laboratory. `http://www.esrl.noaa.gov/psd/`.

32 Joan Feigenbaum, Sampath Kannan, Martin J. Strauss, and Mahesh Viswanathan. An approximate l1-difference algorithm for massive data streams. *SIAM J. Comput.*, 32:131–151, January 2003.

33 Philippe Flajolet and G. Nigel Martin. Probabilistic counting. In *FOCS*, pages 76–82. IEEE, 1983.

34 Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1:397–413, August 1993.

35 Anna C. Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In

Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, STOC'02, pages 389–398, New York, NY, USA, 2002.

36 Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD'01, pages 58–66, New York, NY, USA, 2001.

37 Sudipto Guha. Tight results for clustering and summarizing data streams. In *Proceedings of the 12th International Conference on Database Theory*, ICDT'09, pages 268–275, New York, NY, USA, 2009. ACM.

38 Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-streams and histograms. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, STOC'01, pages 471–475, New York, NY, USA, 2001. ACM.

39 Sudipto Guha, Nick Koudas, and Kyuseok Shim. Approximation and streaming algorithms for histogram construction problems. *ACM Trans. Database Syst.*, 31:396–438, March 2006.

40 Sudipto Guha and Kyuseok Shim. A note on linear time algorithms for maximum error histograms. *IEEE Trans. on Knowl. and Data Eng.*, 19:993–997, July 2007.

41 Sudipto Guha, Kyuseok Shim, and Jungchul Woo. Rehist: Relative error histogram construction algorithms. In *Very Large Data Bases*, pages 300–311, 2004.

42 Monika R. Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. *External memory algorithms*, pages 107–118, 1999.

43 Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 53:307–323, May 2006.

44 Piotr Indyk and David Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC'05, pages 202–208, New York, NY, USA, 2005.

45 C. L. Mallows J. M. Chambers and B. W. Stuck. A method for simulating stable random variables. *Journal of the American Statistical Association*, 71:340–344, June 1976.

46 H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB'98, pages 275–286, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

47 Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS'10, pages 41–52, New York, NY, USA, 2010. ACM.

48 Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, March 2003.

49 Panagiotis Karras, Dimitris Sacharidis, and Nikos Mamoulis. Exploiting duality in summarization with deterministic guarantees. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD'07, pages 380–389, New York, NY, USA, 2007. ACM.

50 Eyal Kushilevitz and Noam Nisan. *Communication complexity.* Cambridge University Press, 1997.

51 LHC Computing Grid. `http://lcg.web.cern.ch/LCG/`.

52 Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB'02, pages 346–357. VLDB Endowment, 2002.

53 Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings of the 1998 ACM*

SIGMOD international conference on Management of data, SIGMOD'98, pages 426–435, New York, NY, USA, 1998.

54 Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD'99, pages 251–262, New York, NY, USA, 1999.

55 Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, New York, NY, USA, 2005.

56 Ketan Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, 1993.

57 M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In Haran Boral and Per-Åke Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, pages 28–36. ACM Press, 1988.

58 S. Muthukrishnan. Data streams: algorithms and applications. *Found. Trends Theor. Comput. Sci.*, 1(2):117–236, 2005.

59 Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD'84, pages 256–276, New York, NY, USA, 1984.

60 Nicole Schweikardt. One-pass algorithm. In Ling Liu and M. Tamer Zsu, editors, *Encyclopedia of Database Systems*, pages 1948–1949. Springer Publishing Company, Incorporated, 1st edition, 2009.

61 P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD'79, pages 23–34, New York, NY, USA, 1979. ACM.

62 Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys'04, pages 239–249, New York, NY, USA, 2004.

63 Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11:37–57, March 1985.

64 Mark N. Wegman and J. Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265 – 279, 1981.

65 David Woodruff. Optimal space lower bounds for all frequency moments. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA'04, pages 167–175, Philadelphia, PA, USA, 2004.