CHAPTER 8

Algorithms for Data Streams

CAMIL DEMETRESCU and IRENE FINOCCHI

8.1 INTRODUCTION

Efficient processing over massive data sets has taken an increased importance in the last few decades due to the growing availability of large volumes of data in a variety of applications in computational sciences. In particular, monitoring huge and rapidly changing streams of data that arrive online has emerged as an important data management problem: Relevant applications include analyzing network traffic, online auctions, transaction logs, telephone call records, automated bank machine operations, and atmospheric and astronomical events. For these reasons, the streaming model has recently received a lot of attention. This model differs from computation over traditional stored data sets since algorithms must process their input by making one or a small number of passes over it, using only a limited amount of working memory. The streaming model applies to settings where the size of the input far exceeds the size of the main memory available and the only feasible access to the data is by making one or more passes over it.

Typical streaming algorithms use space at most polylogarithmic in the length of the input stream and must have fast update and query times. Using sublinear space motivates the design for summary data structures with small memory footprints, also known as synopses [34]. Queries are answered using information provided by these synopses, and it may be impossible to produce an exact answer. The challenge is thus to produce high quality approximate answers, that is, answers with confidence bounds on the possible error: Accuracy guarantees are typically made in terms of a pair of user-specified parameters, ε and δ, meaning that the error in answering a query is within a factor of $1 + \varepsilon$ of the true answer with probability at least $1 - \delta$. The space and update time will depend on these parameters and the goal is to limit this dependence as much as possible.

Major progress has been achieved in the last 10 years in the design of streaming algorithms for several fundamental data sketching and statistics problems, for which several different synopses have been proposed. Examples include number of distinct

Handbook of Applied Algorithms: Solving Scientific, Engineering and Practical Problems
Edited by Amiya Nayak and Ivan Stojmenović

items, frequency moments, L_1 and L_2 norms of vectors, inner products, frequent items, heavy hitters, quantiles, histograms, and wavelets. Recently, progress has been achieved for other problem classes, including computational geometry (e.g., clustering and minimum spanning trees) and graphs (e.g., triangle counting and spanners). At the same time, there has been a flurry of activity in proving impossibility results, devising interesting lower bound techniques, and establishing important complementary results.

This chapter is intended as an overview of this rapidly evolving area. The chapter is not meant to be comprehensive, but rather aims at providing an outline of the main techniques used for designing algorithms or for proving lower bounds. We refer the interested reader to the works by Babcock et al. [7], Gibbons and Matias [34] and Muthukrishnan [57] for an extensive discussion of problems and results not mentioned here.

8.1.1 Applications

As observed before, the primary application of data stream algorithms is to monitor continuously huge and rapidly changing streams of data in order to support exploratory analyses and to detect correlations, rare events, fraud, intrusion, and unusual or anomalous activities. Such streams of data may be, for example, performance measurements in traffic management, all detail records in telecommunications, transactions in retail chains, ATM operations in banks, bids in online auctions, log records generated by Web Servers, or sensor network data. In all these cases, the volumes of data are huge (several terabytes or even petabytes), and records arrive at a rapid rate. Other relevant applications for data stream processing are related, for example, to processing massive files on secondary storage and to monitoring the contents of large databases or data warehouse environments. In this section, we highlight some typical needs that arise in these contexts.

8.1.1.1 Network Management Perhaps the most prominent application is related to network management. This involves monitoring and configuring network hardware and software to ensure smooth operations. Consider, for example, traffic analysis in the Internet. Here, as IP packets flow through the routers, we would like to monitor link bandwidth usage, to estimate traffic demands, to detect faults, congestion, and usage patterns. Typical queries that we would be able to answer are thus the following. How many IP addresses used a given link in a certain period of time? How many bytes were sent between a pair of IP addresses? Which are the top 100 IP addresses in terms of traffic? What is the average duration of an IP session? Which sessions transmitted more than 1000 bytes? Which IP addresses are involved in more than 1000 sessions? All these queries are heavily motivated by traffic analysis, fraud detection, and security.

To get a rough estimate of the amount of data that need to be analyzed to answer one such query, consider that each router can forward up to 1 billion packets per hour, and each Internet Service Provider may have many hundreds of routers: thus, many terabytes of data per hour need to be processed. These data arrive at a rapid rate, and

we therefore need algorithms to mine patterns, process queries, and compute statistics on such data streams in almost real time.

8.1.1.2 Database Monitoring Many commercial database systems have a query optimizer used for estimating the cost of complex queries. Consider, for example, a large database that undergoes transactions (including updates). Upon the arrival of a complex query q, the optimizer may run some simple queries in order to decide an optimal query plan for q: In particular, a principled choice of an execution plan by the optimizer depends heavily on the availability of statistical summaries such as histograms, the number of distinct values in a column for the tables referenced in a query, or the number of items that satisfy a given predicate. The optimizer uses this information to decide between alternative query plans and to optimize the use of resources in multiprocessor environments. The accuracy of the statistical summaries greatly impacts the ability to generate good plans for complex SQL queries. The summaries, however, must be computed quickly: In particular, examining the entire database is typically regarded as prohibitive.

8.1.1.3 Online Auctions During the last few years, online implementations of auctions have become a reality, thanks to the Internet and to the wide use of computer-mediated communication technologies. In an online auction system, people register to the system, open auctions for individual items at any time, and then submit continuously items for auction and bids for items. Statistical estimation of auction data is thus very important for identifying items of interest to vendors and purchasers, and for analyzing economic trends.

Typical queries may require to convert the prices of incoming bids between different currencies, to select all bids of a specified set of items, to maintain a table of the currently open auctions, to select the items with the most bids in a specified time interval, to maintain the average selling price over the items sold by each seller, to return the highest bid in a given period of time, or to monitor the average closing price (i.e., the price of the maximum bid, or the starting price of the auction in case there were no bids) across items in each category.

8.1.1.4 Sequential Disk Accesses In modern computing platforms, the access times to main memory and disk vary by several orders of magnitude. Hence, when the data reside on disk, it is much more important to minimize the number of I/Os (i.e., the number of disk accesses) than the CPU computation time as it is done in traditional algorithms theory. Many *ad hoc* algorithmic techniques have been proposed in the external memory model for minimizing the number of I/Os during a computation (see, e.g., the work by Vitter [64]).

Due to the high sequential access rates of modern disks, streaming algorithms can also be effectively deployed for processing massive files on secondary storage, providing new insights into the solution of several computational problems in external memory. In many applications managing massive data sets, using secondary and tertiary storage devices is indeed a practical and economical way to store and move data: such large and slow external memories, however, are best optimized for sequential

access, and thus naturally produce huge streams of data that need to be processed in a small number of sequential passes. Typical examples include data access to database systems [39] and analysis of Internet archives stored on tape [43]. The streaming algorithms designed with these applications in mind may have a greater flexibility: Indeed, the rate at which data are processed can be adjusted, data can be processed in chunks, and more powerful processing primitives (e.g., sorting) may be available.

8.1.2 Overview of the Literature

The problem of computing in a small number of passes over the data appears already in papers from the late 1970s. Morris, for instance, addressed the problem of keeping approximate counts of large numbers [55]. Munro and Paterson [56] studied the space required for selection when at most P passes over the data can be performed, giving almost matching upper and lower bounds as a function of P and of the input size. The paper by Alon et al. [5,6], awarded in 2005 with the Gödel Prize for outstanding papers in the area of theoretical computer science, provided the foundations of the field of streaming and sketching algorithms. This seminal work introduced the novel technique of designing small randomized linear projections that allow the approximation (to user specified precision) of the frequency moments of a data set and other quantities of interest. The computation of frequency moments is now fully understood, with almost matching (up to polylogarithmic factors) upper bounds [12,20,47] and lower bounds [9,14,46,62]. Namely, Indyk and Woodruff [47] presented the first algorithm for estimating the kth frequency moment using space $\tilde{O}(n^{1-2/k})$. A simpler one-pass algorithm is described in [12].

Since 1996, many fundamental data statistics problems have been efficiently solved in streaming models. For instance, the computation of frequent items is particularly relevant in network monitoring applications and has been addressed, for example, in many other works [1,16,22,23,51,54]. A plethora of other problems have been studied in the last few years, designing solutions that hinge upon many different and interesting techniques. Among them, we recall sampling, probabilistic counting, combinatorial group testing, core sets, dimensionality reduction, and tree-based methods. We will provide examples of application of some of these techniques in Section 8.3. An extensive bibliography can be found in the work by Muthukrishnan [57]. The development of advanced techniques made it possible to solve progressively more complex problems, including the computation of histograms, quantiles, norms, as well as geometric and graph problems.

Histograms capture the distribution of values in a data set by grouping values into buckets and maintaining suitable summary statistics for each bucket. Different kinds of histograms exist: for example, in an equidepth histogram the number of values falling into each bucket is uniform across all buckets. The problem of computing these histograms is strictly related to the problem of maintaining the quantiles for the data set: quantiles represent indeed the bucket boundaries. These problems have been addressed, for example, in many other works [18,36,37,40,41,56,58,59]. Wavelets are also widely used to provide summarized representations of data: works on computing wavelet coefficients in data stream models include [4,37,38,60].

A few fundamental works consider problems related to norm estimation, for example, dominance norms and L_p sums [21,44]. In particular, Indyk pioneered the design of sketches based on random variables drawn from stable distributions (which are known to exist) and applied this idea to the problem of estimating L_p sums [44].

Geometric problems have also been the subject of much recent research in the streaming model [31,32,45]. In particular, clustering problems received special attention: given a set of points with a distance function defined on them, the goal is to find a clustering solution (a partition into clusters) that optimizes a certain objective function. Classical objective functions include minimizing the sum of distances of points to their closest median (k-median) or minimizing the maximum distance of a point to its closest center (k-center). Streaming algorithms for such problem are presented, for example, in the works by Charikar [17] and Guha et al. [42].

Differently from most data statistics problems, where $O(1)$ passes and polylogarithmic working space have been proven to be enough to find approximate solutions, many classical graph problems seem to be far from being solved within similar bounds: for many classical graph problems, linear lower bounds on the space $\times$ passes product are indeed known [43]. A notable exception is related to counting triangles in graphs, as discussed in the works by Bar-Yossef et al. [10], Buriol et al. [13], and Jowhari and Ghodsi [49]. Some recent papers show that several graph problems can be solved with one or few passes in the semi-streaming model [26–28,53] where the working memory size is $O(n \cdot \text{polylog } n)$ for an input graph with n vertices: in other words, akin to semi-external memory models [2,64] there is enough space to store vertices, but not edges of the graph. Other works, such as [3,25,61], consider the design of streaming algorithms for graph problems when the model allows more powerful primitives for accessing stream data (e.g., use of intermediate temporary streams and sorting).

8.1.3 Chapter Outline

This chapter is organized as follows. In Section 8.2 we describe the most common data stream models: such models differ in the interpretation of the data on the stream (each item can either be a value itself or indicate an update to a value) and in the primitives available for accessing and processing stream items. In Section 8.3 we focus on techniques for proving upper bounds: we describe some mathematical and algorithmic tools that have proven to be useful in the construction of synopsis data structures (including randomization, sampling, hashing, and probabilistic counting) and we first show how these techniques can be applied to classical data statistics problems. We then move to consider graph problems as well as techniques useful in streaming models that provide more powerful primitives for accessing stream data in a nonlocal fashion (e.g., simulations of parallel algorithms). In Section 8.4 we address some lower bound techniques for streaming problems, using the computation of the number of distinct items in a data stream as a running example: we explore the use of reductions of problems in communication complexity to streaming problems,

and we discuss the use of randomization and approximation in the design of efficient synopses. In Section 8.5 we summarize our contribution.

8.2 DATA STREAM MODELS

A variety of models exist for data stream processing: the differences depend on how stream data should be interpreted and which primitives are available for accessing stream items. In this section we overview the main features of the most commonly used models.

8.2.1 Classical Streaming

In *classical data streaming* [5,43,56,57], input data are accessed sequentially in the form of a data stream $\Sigma = x_1, ..., x_n$ and need to be processed using a working memory that is small compared to the length n of the stream. The main parameters of the model are the number p of sequential passes over the data, the size s of the working memory, and the per-item processing time. All of them should be kept small: typically, one strives for one pass and polylogarithmic space, but this is not a requirement of the model.

There exist at least three variants of classical streaming, dubbed (in increasing order of generality) *time series*, *cash register*, and *turnstile* [57]. Indeed, we can think of stream items $x_1, ..., x_n$ as describing an underlying signal A, that is, a one-dimensional function over the reals. In the time series model, each stream item x_i represents the ith value of the underlying signal, that is, $x_i = A[i]$. In the other models, each stream item x_i represents an update of the signal: namely, x_i can be thought of as a pair (j, U_i), meaning that the jth value of the underlying signal must be changed by the quantity U_i, that is, $A_i[j] = A_{i-1}[j] + U_i$. The partially dynamic scenario in which the signal can be only incremented, that is, $U_i \geq 0$, corresponds to the cash register model, while the fully dynamic case yields the turnstile model.

8.2.2 Semi-Streaming

Despite the heavy restrictions of classical data streaming, we will see in Section 8.3 that major success has been achieved for several data sketching and statistics problems, where $O(1)$ passes and polylogarithmic working space have been proven to be enough to find approximate solutions. On the contrary, there exist many natural problems (including most problems on graphs) for which linear lower bounds on $p \times s$ are known, even using randomization and approximation: these problems cannot be thus solved within similar polylogarithmic bounds. Some recent papers [27,28,53] have therefore relaxed the polylog space requirements considering a *semi-streaming* model, where the working memory size is $O(n \cdot \text{polylog } n)$ for an input graph with n vertices: in other words, akin to semi-external memory models [2,64], there is enough space to store vertices, but not edges of the graph. We will see in Section 8.3.3 that some complex graph problems can be solved in semi-streaming, including spanners, matching, and diameter estimation.

8.2.3 Streaming with a Sorting Primitive

Motivated by technological factors, some authors have recently started to investigate the computational power of even less restrictive streaming models. Today's computing platforms are equipped with large and inexpensive disks highly optimized for sequential read/write access to data, and among the primitives that can efficiently access data in a nonlocal fashion, sorting is perhaps the most optimized and well understood. These considerations have led to introduce the *stream-sort* model [3,61]. This model extends classical streaming in two ways: the ability to write intermediate temporary streams and the ability to reorder them at each pass for free. A stream-sort algorithm alternates streaming and sorting passes: a streaming pass, while reading data from the input stream and processing them in the working memory, produces items that are sequentially appended to an output stream; a sorting pass consists of reordering the input stream according to some (global) partial order and producing the sorted stream as output. Streams are pipelined in such a way that the output stream produced during pass i is used as input stream at pass $i + 1$. We will see in Section 8.3.4 that the combined use of intermediate temporary streams and of a sorting primitive yields enough power to solve efficiently (within polylogarithmic passes and memory) a variety of graph problems that cannot be solved in classical streaming. Even without sorting, the model is powerful enough for achieving space–passes trade-offs [25] for graph problems for which no sublinear memory algorithm is known in classical streaming.

8.3 ALGORITHM DESIGN TECHNIQUES

Since data streams are potentially unbounded in size, when the amount of computation memory is bounded it may be impossible to produce an exact answer. In this case, the challenge is to produce high quality approximate answers, that is, answers with confidence bounds on the possible error. The typical approach is to maintain a "lossy" summary of the data stream by building up a *synopsis data structure* with memory footprint substantially smaller than the length of the stream. In this section we describe some mathematical and algorithmic techniques that have proven to be useful in the construction of such synopsis data structures. Besides the ones considered in this chapter, many other interesting techniques have been proposed: the interested reader can find pointers to relevant works in Section 8.1.2. Rather than being comprehensive, our aim is to present a small amount of results in sufficient detail that the reader can get a feeling of some common techniques used in the field.

The most natural approach to designing streaming algorithms is perhaps to maintain a small *sample* of the data stream: if the sample captures well the essential characteristics of the entire data set with respect to a specific problem, evaluating a query over the sample may provide reliable approximation guarantees for that problem. In Section 8.3.1 we discuss how to maintain a bounded size sample of a (possibly unbounded) data stream and describe applications of sampling to the problem of finding frequent items in a data stream.

Useful randomized synopses can also be constructed hinging upon hashing techniques. In Section 8.3.2 we address the design of *hash-based sketches* for estimating the number of distinct items in a data stream. We also discuss the main ideas behind the design of randomized sketches for the more general problem of estimating the frequency moments of a data set: the seminal paper by Alon et al. [5] introduced the technique of designing small randomized linear projections that summarize large amounts of data and allow frequency moments and other quantities of interest to be approximated to user-specified precision. As quoted from the Gödel Award Prize ceremony, this paper "set the pattern for a rapidly growing body of work, both theoretical and applied, creating the now burgeoning fields of streaming and sketching algorithms."

Sections 8.3.3 and 8.3.4 are mainly devoted to the semi-streaming and stream-sort models. In Section 8.3.3 we focus on techniques that can be applied to solve complex graph problems in $O(1)$ passes and $\tilde{O}(n)$ space. In Section 8.3.4, finally, we analyze the use of more powerful primitives for accessing stream data, showing that sorting yields enough power to solve efficiently a variety of problems for which efficient solutions in classical streaming cannot be achieved.

8.3.1 Sampling

A small random sample S of the data often captures certain characteristics of the entire data set. If this is the case, the sample can be maintained in memory and queries can be answered over the sample. In order to use sampling techniques in a data stream context, we first need to address the problem of maintaining a sample of a specified size over a possibly unbounded stream of data that arrive online. Note that simple coin tossing is not possible in streaming applications, as the sample size would be unbounded. The standard solution is to use Vitter's *reservoir sampling* [63] that we describe in the following Sections.

8.3.1.1 Reservoir Sampling This technique dates back to the 1980s [63]. Given a stream Σ of n items that arrive online, at any instant of time reservoir sampling guarantees to maintain a uniform random sample S of fixed size m of the part of stream observed up to that time. Let us first consider the following natural sampling procedure.

> At the beginning, add to S the first m items of the stream. Upon seeing the stream item x_t at time t, add x_t to S with probability m/t. If x_t is added, evict a random item from S (other than x_t).

It is easy to see that at each time $|S| = m$ as desired. The next theorem proves that, at each time, S is actually a uniform random sample of the stream observed so far.

Theorem 1 [[63]] *Let S be a sample of size m maintained over a stream $\Sigma = x_1, \ldots, x_n$ by the above algorithm. Then, at any time t and for each $i \leq t$, the probability that $x_i \in S$ is m/t.*

Proof. We use induction on t. The base step is trivial. Let us thus assume that the claim is true up to time; t that is, by inductive hypothesis $\Pr[x_i \in S] = m/t$ for each $i \leq t$. We now examine how S can change at time $t+1$, when item x_{t+1} is considered for addition. Consider any item x_i with $i < t+1$. If x_{t+1} is not added to S (this happens with probability $1 - m/(t+1)$), then x_i has the same probability of being in S of the previous step (i.e., m/t). If x_{t+1} is added to S (this happens with probability $m/(t+1)$), then x_i has a probability of being in S equal to $(m/t)(1-1/m)$, since it must have been in S at the previous step and must not be evicted at the current step. Thus, for each $i \leq t$, at time $t+1$ we have

$$\Pr[x_i \in S] = \left(1 - \frac{m}{t+1}\right)\frac{m}{t} + \frac{m}{t+1}\left[\frac{m}{t}\left(1 - \frac{1}{m}\right)\right] = \frac{m}{t+1}.$$

The fact that x_{t+1} is added to S with probability $m/(t+1)$ concludes the proof. □

Instead of flipping a coin for each element (that requires to generate n random values), the reservoir sampling algorithm randomly generates the number of elements to be skipped before the next element is added to S. Special care is taken to generate these skip numbers, so as to guarantee the same properties that we discussed in Theorem 1 for the naïve coin-tossing approach. The implementation based on skip numbers has the advantage that the number of random values to be generated is the same as the number of updates of the sample S. We refer to the work by Vitter [63] for the details and the analysis of this implementation.

We remark that reservoir sampling works well for insert and updates of the incoming data, but runs into difficulties if the data contain deletions. In many applications, however, the timeliness of data is important, since outdated items expire and should be no longer used when answering queries. Other sampling techniques have been proposed that address this issue: see, for example, [8,35,52] and the references therein. Another limitation of reservoir sampling derives from the fact that the stream may contain duplicates, and any value occurring frequently in the sample is a wasteful use of the available space: concise sampling overcomes this limitation representing elements in the sample by pairs (value, count). As described by Gibbons and Matias [33], this natural idea can be used to compress the samples and allows it to solve, for example, the top-k problem, where the k most frequent items need to be identified.

In the rest of this section, we provide a concrete example of how sampling can be effectively applied to certain nontrivial streaming problems. However, as we will see in Section 8.4, there also exist classes of problems for which sampling-based approaches are not effective, unless using a prohibitive (almost linear) amount of memory.

8.3.1.2 An Application of Sampling: Frequent Items Following an approach proposed by Manku and Motwani [51], we will now show how to use sampling to address the problem of identifying frequent items in a data stream, that is, items whose frequency exceeds a user-specified threshold. Intuitively, it should be possible to estimate frequent items by a good sample. The algorithm that we discuss, dubbed

sticky sampling [51], supports this intuition. The algorithm accepts two user-specified thresholds: a frequency threshold $\varphi \in (0, 1)$, and an error parameter $\varepsilon \in (0, 1)$ such that $\varepsilon < \varphi$. Let Σ be a stream of n items $x_1, ..., x_n$. The goal is to report

- all the items whose frequency is at least φn (i.e., there must be no *false negatives*);
- no item with frequency smaller than $(\varphi - \varepsilon)n$.

We will denote by $f(x)$ the true frequency of an item x, and by $f_e(x)$ the frequency estimated by sticky sampling. The algorithm also guarantees small error in individual frequencies; that is, the estimated frequency is less than the true frequency by at most εn. The algorithm is randomized, and in order to meet the two goals with probability at least $1 - \delta$, for a user-specified probability of failure $\delta \in (0, 1)$, it maintains a sample with expected size $2\varepsilon^{-1} \log(\varphi^{-1}\delta^{-1}) = 2t$. Note that the space is independent of the stream length n.

The sample $\mathcal{S}$ is a set of pairs of the form $(x, f_e(x))$. In order to handle potentially unbounded streams, the sampling rate r is not fixed, but is adjusted so that the probability $1/r$ of sampling a stream item decreases as more and more items are considered. Initially, $\mathcal{S}$ is empty and $r = 1$. For each stream item x, if $x \in \mathcal{S}$, then $f_e(x)$ is increased by 1. Otherwise, x is sampled with rate r, that is, with probability $1/r$: if x is sampled, the pair $(x, 1)$ is added to $\mathcal{S}$, otherwise we ignore x and move to the next stream item.

After sampling with rate $r = 1$ the first $2t$ items, the sampling rate increases geometrically as follows: the next $2t$ items are sampled with rate $r = 2$, the next $4t$ items with rate $r = 4$, the next $8t$ items with rate $r = 8$, and so on. Whenever the sampling rate changes, the estimated frequencies of sample items are adjusted so as to keep them consistent with the new sampling rate: for each $(x, f_e(x)) \in \mathcal{S}$, we repeatedly toss an unbiased coin until the coin toss is successful, decreasing $f_e(x)$ by 1 for each unsuccessful toss. We evict $(x, f_e(x))$ from $\mathcal{S}$ if $f_e(x)$ becomes 0 during this process. Effectively, after each sampling rate doubling, $\mathcal{S}$ is transformed to exactly the state it would have been in, if the new rate had been used from the beginning.

Upon a frequency items query, the algorithm returns all sample items whose estimated frequency is at least $(\varphi - \varepsilon)n$.

The following technical lemma will be useful in the analysis of sticky sampling. Although pretty straightforward, we report the proof for the sake of completeness.

Lemma 1 *Let $r \geq 2$ and let n be the number of stream items considered when the sampling rate is r. Then $1/r \geq t/n$, where $t = \varepsilon^{-1} \log(\varphi^{-1}\delta^{-1})$.*

Proof. It can be easily proved by induction on r that $n = rt$ at the beginning of the phase in which sampling rate r is used. The base step, for $r = 2$, is trivial: at the beginning $\mathcal{S}$ contains exactly $2t$ elements by construction. During the phase with sampling rate r, as far as the algorithm works, rt new stream elements are considered; thus, when the sampling rate doubles at the end of the phase, we have $n = 2rt$, as needed to prove the induction step. This implies that during any phase it must be $n \geq rt$, which proves the claim. □

We can now prove that sticky sampling meets the goals in the definition of the frequent items problem with probability at least $1 - \delta$ using space independent of n.

Theorem 2 [[51]] *For any $\varepsilon, \varphi, \delta \in (0, 1)$, with $\varepsilon < \varphi$, sticky sampling solves the frequent items problems with probability at least $1 - \delta$ using a sample of expected size $(2/\varepsilon)\log(\varphi^{-1}\delta^{-1})$.*

Proof. We first note that the estimated frequency of a sample element x is an underestimate of the true frequency, that is, $f_e(x) \leq f(x)$. Thus, if the true frequency is smaller than $(\varphi - \varepsilon)n$, the algorithm will not return x, since it must also be $f_e(x) < (\varphi - \varepsilon)n$.

We now prove that there are no false negatives with probability $\geq 1 - \delta$. Let k be the number of elements with frequency at least φ, and let $y_1, ..., y_k$ be those elements. Clearly, it must be $k \leq 1/\varphi$. There are no false negatives if and only if all the elements $y_1, ..., y_k$ are returned by the algorithm. We now study the probability of the complementary event, proving that it is upper bounded by δ.

$$\Pr[\exists \text{ false negative}] \leq \sum_{i=1}^{k} \Pr[y_i \text{ is not returned}] = \sum_{i=1}^{k} \Pr[f_e(y_i) < (\varphi - \varepsilon)n].$$

Since $f(y_i) \geq \varphi n$ by definition of y_i, we have $f_e(y_i) < (\varphi - \varepsilon)n$ if and only if the estimated frequency of y_i is underestimated by at least ϵn. Any error in the estimated frequency of an element corresponds to a sequence of unsuccessful coin tosses during the first occurrences of the element. The length of this sequence exceeds εn with probability

$$\left(1 - \frac{1}{r}\right)^{\varepsilon n} \leq \left(1 - \frac{t}{n}\right)^{\varepsilon n} \leq e^{-t\varepsilon},$$

where the first inequality follows from Lemma 1. Hence,

$$\Pr[\exists \text{ false negative}] \leq k\, e^{-t\varepsilon} \leq \frac{e^{-t\varepsilon}}{\varphi} = \delta$$

by definition of t. This proves that the algorithm is correct with probability $\geq 1 - \delta$.

It remains to discuss the space usage. The number of stream elements considered at the end of the phase in which sampling rate r is used must be at most $2rt$ (see the proof of Lemma 1 for details). The algorithm behaves as if each element was sampled with probability $1/r$: the expected number of sampled elements is therefore $2t$. □

Manku and Motwani also provide a deterministic algorithm for estimating frequent items: this algorithm guarantees no false negatives and returns no false positives with true frequency smaller than $(\varphi - \varepsilon)n$ [51]. However, the price paid for being deterministic is that the space usage increases to $O((1/\varepsilon)\log(\varepsilon n))$. Other works that describe different techniques for tracking frequent items are, for example, [1,16,22,23,54].

8.3.2 Sketches

In this section we exemplify the use of sketches as randomized estimators of the frequency moments of a data stream. Let $\Sigma = x_1, ..., x_n$ be a stream of n values taken from a universe U of size u, and let f_i, for $i \in U$, be the frequency (number of occurrences) of value i in Σ, that is, $f_i = |\{j : x_j = i\}|$. The kth frequency moment F_k of Σ is defined as

$$F_k = \sum_{i \in U} f_i^k.$$

Frequency moments represent useful statistical information on a data set and are widely used in database applications. In particular, F_0 and F_1 represent the number of distinct values in the data stream and the length of the stream, respectively. F_2, also known as Gini's index, provides valuable information about the skew of the data. F_∞, finally, is related to the maximum frequency element in the data stream, that is, $\max_{i \in U} f_i$.

8.3.2.1 Probabilistic Counting We begin our discussion from the estimation of F_0. The problem of counting the number of distinct values in a data set using small space has been studied since the early 1980s by Flajolet and Martin [29,30], who proposed a hash-based *probabilistic counter*. We first note that a naïve approach to compute the exact value of F_0 would use a counter $c(i)$ for each value i of the universe U, and would therefore require $O(1)$ processing time per item, but linear space. The probabilistic counter of Flajolet and Martin [29,30] relies on hash functions to find a good approximation of F_0 using only $O(\log u)$ bits of memory, where u is the size of the universe U.

The counter consists of an array C of $\log u$ bits. Each stream item is mapped to one of the $\log u$ bits by means of the combination of two functions h and t. The hash function $h : U \to [0, u-1]$ is drawn from a set of strongly 2-universal hash functions: it transforms values of the universe into integers sufficiently uniformly distributed over the set of binary strings of length $\log u$. The function t, for any integer i, gives the number $t(i)$ of trailing zeros in the binary representation of i. Updates and queries work as follows:

- *Counter update*: Upon seeing a stream value x, set $C[t(h(x))]$ to 1.

- *Distinct values query*: Let R be the position of the rightmost 1 in the counter C, with $1 \leq R \leq \log u$. Return 2^R.

Notice that all stream items by the same value will repeatedly set the same counter bit to 1. Intuitively, the fact that h distributes items uniformly over $[0, u-1]$ and the use of function t guarantee that counter bits are selected in accordance with a geometric distribution; that is, $1/2$ of the universe items will be mapped to the first counter bit, $1/4$ will be mapped to the second counter bit, and so on. Thus, it seems reasonable to expect that the first $\log F_0$ counter bits will be set to 1 when the stream contains

F_0 distinct items: this suggests that R, as defined above, yields a good approximation for F_0. We will now give a more formal analysis. We will denote by Z_j the number of distinct stream items that are mapped (by the composition of functions t and h) to a position $\geq j$. Thus, R is the maximum j such that $Z_j > 0$.

Lemma 2 *Let Z_j be the number of distinct stream items x for which $t(h(x)) \geq j$. Then, $E[Z_j] = F_0/2^j$ and $\mathrm{Var}[Z_j] < E[Z_j]$.*

Proof. Let W_x be an indicator random variable whose value is 1 if and only if $t(h(x)) \geq j$. Then, by definition of Z_j,

$$Z_j = \sum_{x \in U \cap \Sigma} W_x. \tag{8.1}$$

Note that $|U \cap \Sigma| = F_0$. We now study the probability that $W_x = 1$. It is not difficult to see that the number of binary strings of length $\log u$ that have exactly j trailing zeros, for $0 \leq j < \log u$, is $2^{\log u-(j+1)}$. Thus, the number of strings that have at least j trailing zeros is $1 + \sum_{i=j}^{\log u-1} 2^{\log u-(i+1)} = 2^{\log u-j}$. Since h distributes items uniformly over $[0, u-1]$, we have that

$$\Pr[W_x = 1] = \Pr[t(h(x)) \geq j] = \frac{2^{\log u-j}}{u} = 2^{-j}.$$

Hence, $E[W_x] = 2^{-j}$ and $\mathrm{Var}[W_x] = E[W_x^2] - E[W_x]^2 = 2^{-j} - 2^{-2j} = 2^{-j}(1 - 2^{-j})$. We are now ready to compute $E[Z_j]$ and $\mathrm{Var}[Z_j]$. By (8.1) and by linearity of expectation we have

$$E[Z_j] = F_0 \cdot \left(1 \cdot \frac{1}{2^j} + 0 \cdot \left(1 - \frac{1}{2^j}\right)\right) = \frac{F_0}{2^j}.$$

Due to pairwise independence (guaranteed by the choice of the hash function h) we have $\mathrm{Var}[W_x + W_y] = \mathrm{Var}[W_x] + \mathrm{Var}[W_y]$ for any $x, y \in U \cap \Sigma$ and thus

$$\mathrm{Var}[Z_j] = \sum_{x \in U \cap \Sigma} \mathrm{Var}[W_x] = \frac{F_0}{2^j}\left(1 - \frac{1}{2^j}\right) < F_0 2^j = E[Z_j].$$

This concludes the proof. □

Theorem 3 *[[5,29,30]] Let F_0 be the exact number of distinct values and let 2^R be the output of the probabilistic counter to a distinct values query. For any $c > 2$, the probability that 2^R is not between F_0/c and $c\,F_0$ is at most $2/c$.*

Proof. Let us first study the probability that the algorithm overestimates F_0 by a factor of c. We begin by noticing that Z_j takes only nonnegative values, and thus we

can apply Markov's inequality to estimate the probability that $Z_j \geq 1$, obtaining

$$\Pr[Z_j \geq 1] \leq \frac{E[Z_j]}{1} = \frac{F_0}{2^j}, \tag{8.2}$$

where the equality is by Lemma 2. If the algorithm overestimates F_0 by a factor of c, then it must exist an index j such that $C[j] = 1$ and $2^j/F_0 > c$ (i.e., $j > \log_2(c\,F_0)$). By definition of Z_j, this implies $Z_{\log_2(c\,F_0)} \geq 1$. Thus,

$$\Pr[\exists j :\ C[j] = 1 \text{ and } 2^j/F_0 > c\,] \leq \Pr[\,Z_{\log_2(c\,F_0)} \geq 1\,] \leq \frac{F_0}{2^{\log_2(c\,F_0)}} = \frac{1}{c},$$

where the last inequality follows from (8.2). The probability that the algorithm overestimates F_0 by a factor of c is therefore at most $1/c$.

Let us now study the probability that the algorithm underestimates F_0 by a factor of $1/c$. Symmetrically to the previous case, we begin by estimating the probability that $Z_j = 0$. Since Z_j takes only nonnegative values, we have

$$\Pr[\,Z_j = 0\,] = \Pr[\,|Z_j - E[Z_j]| \geq E[Z_j]\,] \leq \frac{\mathrm{Var}[Z_j]}{E[Z_j]^2} < \frac{1}{E[Z_j]} = \frac{2^j}{F_0} \tag{8.3}$$

using Chebyshev inequality and Lemma 2. If the algorithm underestimates F_0 by a factor of $1/c$, then there must exist an index j such that $2^j < F_0/c$ (i.e., $j < \log_2(F_0/c)$) and $C[p] = 0$ for all positions $p \geq j$. By definition of Z_j, this implies $Z_{\log_2(F_0/c)} = 0$, and with reasonings similar to the previous case and by using (8.3), we obtain that the probability that the algorithm underestimates F_0 by a factor of $1/c$ is at most $2^{\log_2(F_0/c)}/F_0 = 1/c$.

The upper bounds on the probabilities of overestimates and underestimates imply that the probability that 2^R is not between F_0/c and $c\,F_0$ is at most $2/c$. □

The probabilistic counter of Flajolet and Martin [29,30] assumes the existence of hash functions with some ideal random properties. This assumption has been more recently relaxed by Alon et al. [5], who adapted the algorithm so as to use simpler linear hash functions. We remark that streaming algorithms for computing a $(1+\varepsilon)$-approximation of the number of distinct items are presented, for example, in the work by Bar-Yossef et al. [11].

8.3.2.2 Randomized Linear Projections and AMS Sketches

We now consider the more general problem of estimating the frequency moments F_k of a data set, for $k \geq 2$, focusing on the seminal work by Alon et al. [5].

In order to estimate F_2, Alon et al. introduced a fundamental technique based on the design of small randomized linear projections that summarize some essential properties of the data set. The basic idea of the sketch designed in the work by Alon et al. [5] for estimating F_2 is to define a random variable whose expected value is F_2, and whose variance is relatively small. We follow the description from the work Alon et al. [4].

The algorithm computes μ random variables $Y_1, ..., Y_\mu$ and outputs their median Y as the estimator for F_2. Each Y_i is in turn the average of α independent, identically distributed random variables X_{ij}, with $1 \leq j \leq \alpha$. The parameters μ and α need to be carefully chosen in order to obtain the desired bounds on space, approximation, and probability of error: such parameters will depend on the approximation guarantee λ and on the error probability δ.

Each X_{ij} is computed as follows. Select at random a hash function ξ mapping the items of the universe U to $\{-1, +1\}$: ξ is selected from a family of 4-wise independent hash functions. Informally, 4-wise independence means that for every four distinct values $u_1, ..., u_4 \in U$ and for every 4-tuple $\varepsilon_1, ..., \varepsilon_4 \in \{-1, +1\}$, exactly $(1/16)$-fraction of the hash functions in the family map u_i to ε_i, for $i = 1, ..., 4$. Given ξ, we define $Z_{ij} = \sum_{u \in U} f_u \xi(u)$ and $X_{ij} = Z_{ij}^2$. Notice that Z_{ij} can be considered as a random linear projection (i.e., an inner product) of the frequency vector of the values in U with the random vector associated with such values by the hash function ξ.

It can be proved that $E[Y] = F_2$ and that, thanks to averaging of the X_{ij}, each Y_i has small variance. Computing Y as the median of Y_i allows it to boost the confidence using standard Chernoff bounds. We refer the interested reader to the work by Alon et al. [5] for a detailed proof. We limit here to formalize the statement of the result proved in the work by Alon et al. [5].

Theorem 4 *[[5]] For every $k \geq 1$, $\lambda > 0$, and $\delta > 0$, there exists a randomized algorithm that computes a number Y that deviates from F_2 by more than λF_2 with probability at most δ. The algorithm uses only*

$$O\left(\frac{\log(1/\delta)}{\lambda^2}(\log u + \log n)\right)$$

memory bits and performs one pass over the data.

Let us now consider the case of F_k, for $k \geq 2$. The basic idea of the sketch designed in the work by Alon et al. [5] is similar to that described above, but each X_{ij} is now computed by sampling the stream Σ as follows: an index $p = p_{ij}$ is chosen uniformly at random in $[1, n]$ and the number r of occurrences of x_p in the stream following position p is computed by keeping a counter. X_{ij} is then defined as $n(r^k - (r - 1)^k)$. We refer the interested reader to the works by Alon et al. [4–6] for a detailed description of this sketch and for the extension to the case where the stream length n is not known. We limit here to formalize the statement of the result proved in the work by Alon et al. [5]:

Theorem 5 *[[5]] For every $k \geq 1$, $\lambda > 0$ and $\delta > 0$, there exists a randomized algorithm that computes a number Y such that Y deviates from F_k by more than λF_k with probability at most δ. The algorithm uses*

$$O\left(\frac{k \log(1/\delta)}{\lambda^2} u^{1-1/k}(\log u + \log n)\right)$$

memory bits and performs only one pass over the data.

Notice that Theorem 5 implies that F_2 can be estimated using $O((\log(1/\delta)/\lambda^2)\sqrt{u}(\log u + \log n))$ memory bits: this is worse by a $\sqrt{u}$ factor than the bound obtained in Theorem 4.

8.3.3 Techniques for Graph Problems

In this section we focus on techniques that can be applied to solve graph problems in the classical streaming and semi-streaming models. In Section 8.3.4 we will consider results obtained in less restrictive models that provide more powerful primitives for accessing stream data in a nonlocal fashion (e.g., stream-sort). Graph problems appear indeed to be difficult in classical streaming, and only few interesting results have been obtained so far. This is in line with the linear lower bounds on the space $\times$ passes product proved in the work by Henzinger et al. [43], even using randomization and approximation.

One problem for which sketches could be successfully designed is counting the number of triangles: if the graphs have certain properties, the algorithm presented in the work by Bar-Yossef et al. [10] uses sublinear space. Recently, Cormode and Muthukrishnan [24] studied three fundamental problems on multigraph degree sequences: estimating frequency moments of degrees, finding the heavy hitter degrees, and computing range sums of degree values. In all cases, their algorithms have space bounds significantly smaller than storing complete information. Due to the lower bounds in the work by Henzinger et al. [43], most work has been done in the semi-streaming model, in which problems such as distances, spanners, matchings, girth, and diameter estimation have been addressed [27,28,53]. In order to exemplify the techniques used in these works, in the rest of this section we focus on one such result, related to computing maximum weight matchings.

8.3.3.1 Approximating Maximum Weight Matchings Given an edge weighted, undirected graph $G(V, E, w)$, the weighted matching problem is to find a matching M^* such that $w(M^*) = \sum_{e \in M^*} w(e)$ is maximized. We recall that edges in a matching are such that no two edges have a common end point. We now present a one-pass semi-streaming algorithm that solves the weighted matching problem with approximation ratio $1/6$; that is, the matching M returned by the algorithm is such that

$$w(M^*) \leq 6\, w(M).$$

The algorithm has been proposed in the work by Feigenbaum et al. [27] and is very simple to describe. Algorithms with better approximation guarantees are described in the work by McGregor [53].

As edges are streamed, a matching M is maintained in main memory. Upon arrival of an edge e, the algorithm considers the set $C \subseteq M$ of matching edges

that share an end point with e. If $w(e) > 2w(C)$, then e is added to M while the edges in C are removed; otherwise ($w(e) \leq 2w(C)$) e is ignored.

Note that, by definition of matching, the set C of conflicting edges has cardinality at most 2. Furthermore, since any matching consists of at most $n/2$ edges, the space requirement in bits is clearly $O(n \log n)$.

In order to analyze the approximation ratio, we will use the following notion of replacement tree associated with a matching edge (see also Fig. 8.1). Let e be an edge that belongs to M at the end of the algorithm's execution: the nodes of its replacement tree T_e are edges of graph G, and e is the root of T_e. When e has been added to M, it may have replaced one or two other edges e_1 and e_2 that were previously in M: e_1 and e_2 are children of e in T_e, which can be fully constructed by applying the reasoning recursively. It is easy to upper bound the total weight of nodes of each replacement tree.

Lemma 3 *Let $R(e)$ be the set of nodes of the replacement tree T_e, except for the root e. Then, $w(R(e)) \leq w(e)$.*

Proof. The proof is by induction. When e is a leaf in T_e (base step), $R(e)$ is empty and $w(R(e)) = 0$. Let us now assume that e_1 and e_2 are the children of e in T_e (the case of a unique child is similar). By inductive hypothesis, $w(e_1) \geq w(R(e_1))$ and $w(e_2) \geq w(R(e_2))$. Since e replaced e_1 and e_2, it must have been $w(e) \geq 2\,(w(e_1) + w(e_2))$. Hence, $w(e) \geq w(e_1) + w(e_2) + w(R(e_1)) + w(R(e_2)) = w(R(e))$. □

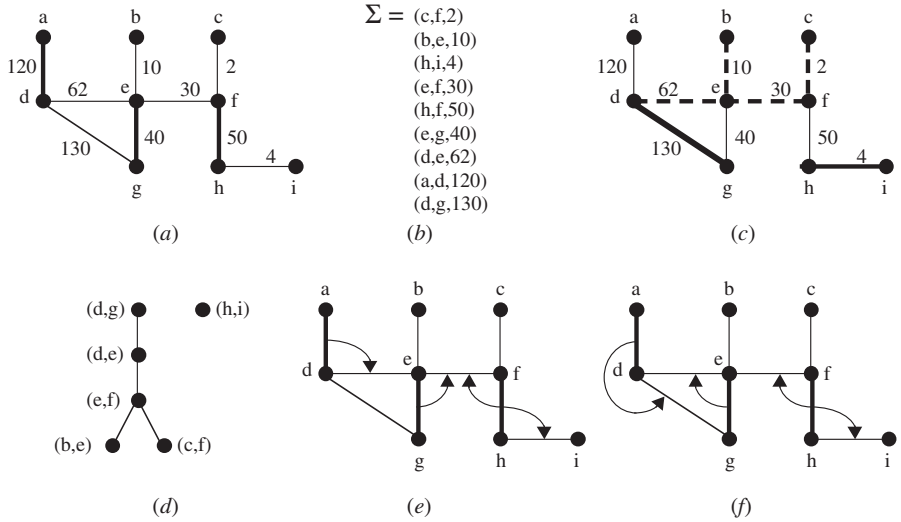

FIGURE 8.1 (*a*) A weighted graph and an optimal matching OPT (bold edges); (*b*) order in which edges are streamed; (*c*) matching M computed by the algorithm (bold solid edges) and edges in the history $H \setminus M$ (dashed edges); (*d*) replacement trees of edges in M; (*e*) initial charging of the weights of edges in OPT; (*f*) charging after the redistribution.

Theorem 6 *[[27]] In one pass and space $O(n \log n)$, the above algorithm constructs a $(1/6)$-approximate weighted matching M.*

Proof. Let $\text{OPT} = \{o_1, o_2, ...\}$ be the set of edges in a maximum weight matching and let $H = \bigcup_{e \in M}(R(e) \cup \{e\})$ be the set of edges that have been part of the matching at some point during the algorithm's execution (these are the nodes of the replacement trees).

We will show an accounting scheme that charges the weight of edges in OPT to edges in H. The charging strategy, for each edge $o \in$OPT, is the following:

- If $o \in H$, we charge $w(o)$ to o itself.
- If $o \notin H$, let us consider the time when o was examined for insertion in M, and let C be the set of edges that share an end point with o and were in M at that time. Since o was not inserted, it must have been $|C| \geq 1$ and $w(o) \leq 2\, w(C)$. If C contains only one edge, we charge $w(o)$ to that edge. If C contains two edges e_1 and e_2, we charge $w(o)w(e_1)/(w(e_1) + w(e_2)) \leq 2\, w(e_1)$ to e_1 and $w(o)w(e_2)/(w(e_1) + w(e_2)) \leq 2\, w(e_2)$ to e_2.

The following two properties hold: (a) the charge of o to any edge e is at most $2\, w(e)$; (b) any edge of H is charged by at most two edges of OPT, one per end point (see also Fig. 8.1).

We now redistribute some charges as follows: if an edge $o \in$ OPT charges an edge $e \in H$ and e gets replaced at some point by an edge $e' \in H$ that also shares an end point with o, we transfer the charge of o from e to e'. With this procedure, property (a) remains valid since $w(e') \geq w(e)$. Moreover, o will always charge an incident edge, and thus property (b) also remains true. In particular, each edge $e \in H \setminus M$ will be now charged by at most one edge in OPT: if at some point there are two edges charging e, the charge of one of them will be transferred to the edge of H that replaced e. Thus, only edges in M can be charged by two edges in OPT. By the above discussion we get

$$w(\text{OPT}) \leq \sum_{e \in H \setminus M} 2w(e) + \sum_{e \in M} 4w(e) = \sum_{e \in M} 2w(R(e)) + \sum_{e \in M} 4w(e)$$

$$\leq \sum_{e \in M} 6w(e) = 6w(M),$$

where the first equality is by definition of H and the last inequality is by Lemma 3. □

8.3.4 Simulation of PRAM Algorithms

In this section we show that a variety of problems for which efficient solutions in classical streaming are not known or impossible to obtain can be solved very efficiently in the stream-sort model discussed in Section 8.2.3. In particular, we show that parallel algorithms designed in the PRAM model [48] can yield very efficient algorithms in the stream-sort model. This technique is very similar to previous methods

developed in the context of external memory management for deriving I/O efficient algorithms (see, e.g., the work by Chiang et al. [19]). We recall that the PRAM is a popular model of parallel computation: it consists of a number of processors (each processor is a standard Random Access Machine) that communicate through a common, shared memory. The computation proceeds in synchronized steps: no processor will proceed with instruction $i + 1$ before all other processors complete the ith step.

Theorem 7 *Let A be a PRAM algorithm that uses N processors and runs in time T. Then, A can be simulated in stream-sort in $p = O(T)$ passes and space $s = O(\log N)$.*

Proof. Let $\Sigma = (1, \text{val}_1)(2, \text{val}_2)\cdots(M, \text{val}_M)$ be the input stream that represents the memory image given as input to algorithm A, where val_j is the value contained at address j, and $M = O(N)$. At each step of algorithm A, processor p_i reads one memory cell at address in_i, updates its internal state st_i, and possibly writes one output cell at address out_i. In a preprocessing pass, we append to Σ the N tuples:

$$(p_1, \text{in}_1, \text{st}_1, \text{out}_1)\cdots(p_N, \text{in}_N, \text{st}_N, \text{out}_N),$$

where in_i and out_i are the cells read and written by p_i at the first step of algorithm A, respectively, and st_i is the initial state of p_i. Each step of A can be simulated by performing the following sorting and scanning passes:

1. We sort the stream so that each (j, val_j) is immediately followed by tuples $(p_i, \text{in}_i, \text{st}_i, \text{out}_i)$ such that $\text{in}_i = j$; that is, the stream has the form
   $(1, \text{val}_1)(p_{i_{11}}, 1, \text{st}_{i_{11}}, \text{out}_{i_{11}})(p_{i_{12}}, 1, \text{st}_{i_{12}}, \text{out}_{i_{12}})\cdots$
   $(2, \text{val}_2)(p_{i_{21}}, 2, \text{st}_{i_{21}}, \text{out}_{i_{21}})(p_{i_{22}}, 2, \text{st}_{i_{22}}, \text{out}_{i_{22}})\cdots$
   $\cdots$
   $(M, \text{val}_M)(p_{i_{M1}}, M, \text{st}_{i_{M1}}, \text{out}_{i_{M1}})(p_{i_{M2}}, M, \text{st}_{i_{M2}}, \text{out}_{i_{M2}})\cdots$
   This can be done, for example, by using $2j$ as sorting key for tuples (j, val_j) and $2\text{in}_i + 1$ as sorting key for tuples $(p_i, \text{in}_i, \text{st}_i, \text{out}_i)$.
2. We scan the stream, performing the following operations:
   - If we read (j, val_j), we let currval $= \text{val}_j$ and we write $(j, \text{val}_j, \text{“old”})$ to the output stream.
   - If we read $(p_i, \text{in}_i, \text{st}_i, \text{out}_i)$, we simulate the task performed by processor p_i, observing that the value val_{in_i} that p_i would read from cell in_i is readily available in currval. Then we write to the output stream $(\text{out}_i, \text{res}_i, \text{“new”})$, where res_i is the value that p_i would write at address out_i, and we write tuple $(p_i, \text{in}'_i, \text{st}'_i, \text{out}'_i)$, where in'_i and out'_i are the cells to be read and written at the next step of A, respectively, and st'_i is the new state of processor p_i.
3. Notice that at this point, for each j we have in the stream a triple of the form $(j, \text{val}_j, \text{“old”})$, which contains the value of cell j before the parallel step, and possibly one or more triples $(j, \text{res}_i, \text{“new”})$, which store the values written by processors to cell j during that step. If there is no “new” value for cell j, we simply drop the “old” tag from $(j, \text{val}_j, \text{“old”})$. Otherwise, we keep for cell j

one of the new triples pruned of the "new" tag, and get rid of the other triples. This can be easily done with one sorting pass, which lets triples by the same j be consecutive, followed by one scanning pass, which removes tags and duplicates.

To conclude the proof, we observe that if A performs T steps, then our stream-sort simulation requires $p = O(T)$ passes. Furthermore, the number of bits of working memory required to perform each processor task simulation and to store currval is $s = O(\log N)$. □

Theorem 7 provides a systematic way of constructing streaming algorithms (in the stream-sort model) for several fundamental problems. Prominent examples are list ranking, Euler tour, graph connectivity, minimum spanning tree, biconnected components, and maximal independent set, among others: for these problems there exist parallel algorithms that use a polynomial number of processors and polylogarithmic time (see, e.g., the work by Jájá [48]). Hence, according to Theorem 7, these problems can be solved in the stream-sort model within polylogarithmic space and passes. Such bounds essentially match the results obtainable in more powerful computational models for massive data sets, such as the parallel disk model [64]. As observed by Aggarwal et al. [3], this suggests that using more powerful, harder to implement models may not always be justified.

8.4 LOWER BOUNDS

An important technique for proving streaming lower bounds is based on communication complexity lower bounds [43]. A crucial restriction in accessing a data stream is that items are revealed to the algorithm sequentially. Suppose that the solution of a computational problem needs to compare two items directly; one may argue that if the two items are far apart in the stream, one of them must be kept in main memory for long time by the algorithm until the other item is read from the stream. Intuitively, if we have limited space and many distant pairs of items to be compared, then we cannot hope to solve the problem unless we perform many passes over the data. We formalize this argument by showing reductions of communication problems to streaming problems. This allows us to prove lower bounds in streaming based on lower bounds in communication complexity. To illustrate this technique, we prove a lower bound for the element distinctness problem, which clearly implies a lower bound for the computation of the number of distinct items F_0 addressed in Section 8.3.2.

Theorem 8 *Any deterministic or randomized algorithm that decides whether a stream of n items contains any duplicates requires $p = \Omega(n/s)$ passes using s bits of working memory.*

Proof. The proof follows from a two-party communication complexity lower bound for the bit-vector-disjointness problem. In this problem, Alice has an n-bit-vector A and Bob has an n-bit-vector B. They want to know whether $A \cdot B > 0$, that is, whether

there is at least one index $i \in \{1, \ldots, n\}$ such that $A[i] = B[i] = 1$. By a well-known communication complexity lower bound [50], Alice and Bob must communicate $\Omega(n)$ bits to solve the problem. This results holds also for randomized protocols: any algorithm that outputs the correct answer with high probability must communicate $\Omega(n)$ bits.

We now show that bit-vector-disjointness can be reduced to the element distinctness streaming problem. The reduction works as follows. Alice creates a stream of items S_A containing indices i such that $A[i] = 1$. Bob does the same for B, that is, he creates a stream of items S_B containing indices i such that $B[i] = 1$. Alice runs a streaming algorithm for element distinctness on S_A, then she sends the content of her working memory to Bob. Bob continues to run the same streaming algorithm starting from the memory image received from Alice, and reading items from the stream S_B. When the stream is over, Bob sends his memory image back to Alice, who starts a second pass on S_A, and so on. At each pass, they exchange $2s$ bits. At the end of the last pass, the streaming algorithm can answer whether the stream obtained by concatenating S_A and S_B contains any duplicates; since this stream contains duplicates if and only if $A \cdot B > 0$, this gives Alice and Bob a solution to the problem.

Assume by contradiction that the number of passes performed by Alice and Bob over the stream is $o(n/s)$. Since at each pass they communicate $2s$ bits, then the total number of bits sent between them over all passes is $o(n/s) \cdot 2s = o(n)$, which is a contradiction as they must communicate $\Omega(n)$ bits as noticed above. Thus, any algorithm for the element distinctness problem that uses s bits of working memory requires $p = \Omega(n/s)$ passes. □

Lower bounds established in this way are information-theoretic, imposing no restrictions on the computational power of the algorithms. The general idea of reducing a communication complexity problem to a streaming problem is very powerful, and allows it to prove several streaming lower bounds. Those range from computing statistical summary information such as frequency moments [5] to graph problems such as vertex connectivity [43], and imply that for many fundamental problems there are no one-pass exact algorithms with a working memory significantly smaller than the input stream.

A natural question is whether approximation can make a significant difference for those problems, and whether randomization can play any relevant role. An interesting observation is that there are problems, such as the computation of frequency moments, for which neither randomization nor approximation is powerful enough for getting a solution in one pass and sublinear space, unless they are used together.

8.4.1 Randomization

As we have seen in the proof of Theorem 8, lower bounds based on the communication complexity of the bit-vector-disjointness problem hold also for randomized algorithms, which yields clear evidence that randomization without approximation may not help. The result of Theorem 8 can be generalized for all one-pass frequency moments. In particular, it is possible to prove that any randomized algorithm for com-

puting the frequency moments that outputs the correct result with probability higher than $1/2$ in one pass must use $\Omega(n)$ bits of working memory. The theorem can be proven using communication complexity tools.

Theorem 9 *[[6]] For any nonnegative integer $k \neq 1$, any randomized algorithm that makes one pass over a sequence of at least $2n$ items drawn from the universe $U = \{1, 2, \ldots, n\}$ and computes F_k exactly with probability $> 1/2$ must use $\Omega(n)$ bits of working memory.*

8.4.2 Approximation

Conversely, we can show that any deterministic algorithm for computing the frequency moments that approximates the correct result within a constant factor in one pass must use $\Omega(n)$ bits of working memory. Differently from the lower bounds addressed earlier in this section, we give a direct proof of this result without resorting to communication complexity arguments.

Theorem 10 *[[6]] For any nonnegative integer $k \neq 1$, any deterministic algorithm that makes one pass over a sequence of at least $n/2$ items drawn from the universe $U = \{1, 2, \ldots, n\}$ and computes a number Y such that $|Y - F_k| \leq F_k/10$ must use $\Omega(n)$ bits of working memory.*

Proof. The idea of the proof is to show that if the working memory is not large enough, for any deterministic algorithm (which does not use random bits) there exist two subsets S_1 and S_2 in a suitable collection of subsets of U such that the memory image of the algorithm is the same after reading either S_1 or S_2; that is, S_1 and S_2 are indistinguishable. As a consequence, the algorithm has the same memory image after reading either $S_1 : S_1$ or $S_2 : S_1$, where $A : B$ denotes the stream of items that starts with the items of A and ends with the items of B. If S_1 and S_2 have a small intersection, then the two streams $S_1 : S_1$ and $S_2 : S_1$ must have rather different values of F_k, and the algorithm must necessarily make a large error on estimating F_k on at least one of them. We now give more details on the proof assuming that $k \geq 2$. The case $k = 0$ can be treated symmetrically.

Using a standard construction in coding theory, it is possible to build a family $\mathcal{F}$ of $2^{\Omega(n)}$ subsets of U of size $n/4$ each such that any two of them have at most $n/8$ common items. Notice that, for every set in $\mathcal{F}$, the frequency of any value of U in that set is either 0 or 1. Fix a deterministic algorithm and let $s < \log_2 \mathcal{F}$ be the size of its working memory. Since the memory can assume at most 2^s different configurations and we have $|\mathcal{F}| > 2^s$ possible distinct input sets in $\mathcal{F}$, then by the pigeonhole principle there must be two input sets $S_1, S_2 \in \mathcal{F}$ such that the memory image of the algorithm after reading either one of them is the same. Now, if we consider the two streams $S_1 : S_1$ and $S_2 : S_1$, the memory image of the algorithm after processing either one of them is the same. Since by construction of $\mathcal{F}$, S_1 and S_2 contain $n/4$ items each, and have at most $n/8$ items in common, then

- Each of the $n/4$ distinct items in $S_1 : S_1$ has frequency 2, thus

$$F_k^{S_1:S_1} = \sum_{i=1}^{n} f_i^k = 2^k \cdot \frac{n}{4}.$$

- If S_1 and S_2 have exactly $n/8$ items in common, then $S_2 : S_1$ contains exactly $n/8 + n/8 = n/4$ items with frequency 1 and $n/8$ items with frequency 2. Hence,

$$F_k^{S_2:S_1} = \sum_{i=1}^{n} f_i^k = \frac{n}{4} + 2^k \cdot \frac{n}{8}.$$

  Notice that, for $k \geq 2$, $F_k^{S_2:S_1}$ can only decrease as $|S_1 \cap S_2|$ decreases, and therefore we can conclude that

$$F_k^{S_2:S_1} \leq \frac{n}{4} + 2^k \cdot \frac{n}{8}.$$

To simplify the notation, let $A = F_k^{S_2:S_1}$ and $B = F_k^{S_1:S_1}$. The maximum relative error performed by the algorithm on either input $S_2 : S_1$ or input $S_1 : S_1$ is

$$\max\left\{\frac{|Y - A|}{A}, \frac{|Y - B|}{B}\right\}.$$

In order to prove that the maximum relative error is always $\geq 1/10$, it is sufficient to show that

$$\frac{|Y - B|}{B} < \frac{1}{10} \quad \Rightarrow \quad \frac{|Y - A|}{A} \geq \frac{1}{10}. \tag{8.4}$$

Let $C = n/4 + 2^k \cdot n/8$. For $k \geq 2$, it is easy to check that $A \leq C \leq B = 2^k \cdot n/4$. Moreover, the maximum relative error obtained for any $Y < A$ is larger than the maximum relative error obtained for $Y = A$ (similarly for $Y > B$): thus, the value of Y that minimizes the relative error is such that $A \leq Y \leq B$. Under this hypothesis, $|Y - B| = B - Y$ and $|Y - A| = Y - A$. With simple calculations, we can show that proving (8.4) is equivalent to proving that

$$Y > \frac{9}{10}B \quad \Rightarrow \quad Y \geq \frac{11}{10}A.$$

Notice that $C = n/4 + B/2$. Using this fact, it is not difficult to see that $9B \geq 11C$ for any $k \geq 2$, and therefore the above implication is always satisfied since $C \geq A$.

Since the maximum relative error performed by the algorithm on either input $S_1 : S_1$ or input $S_2 : S_1$ is at least $1/10$, we can conclude that if we use fewer than $\log_2 \mathcal{F} = \Omega(n)$ memory bits, there is an input on which the algorithm outputs a value Y such that $|Y - F_k| > F_k/10$, which proves the claim. □

8.4.3 Randomization and Approximation

A natural approach that combines randomization and approximation would be to use random sampling to get an estimator of the solution. Unfortunately, this may not always work: as an example, Charikar et al. [15] have shown that estimators based on random sampling do not yield good results for F_0.

Theorem 11 *[[15]] Let E be a (possibly adaptive and randomized) estimator of F_0 that examines at most r items in a set of n items and let* $\text{err} = \max\{E/F_0, F_0/E\}$ *be the error of the estimator. Then, for any $p > 1/e^r$, there is a choice of the set of items such that* $\text{err} \geq \sqrt{((n-r)/2r)\ln(1/p)}$ *with probability at least p.*

The result of Theorem 11 states that no good estimator can be obtained if we only examine a fraction of the input. On the contrary, as we have seen in Section 8.3.2, hashing techniques that examine all items in the input allow it to estimate F_0 within an arbitrary fixed error bound with high probability using polylogarithmic working memory space for any given data set.

We notice that, while the ideal goal of a streaming algorithm is to solve a problem using a working memory of size polylogarithmic in the size of the input stream, for some problems this is impossible even using approximation and randomization, as shown in the following theorem from the work by Alon et al. [6].

Theorem 12 *[[6]] For any fixed integer $k > 5$, any randomized algorithm that makes one pass over a sequence of at least n items drawn from the universe $U = \{1, 2, \ldots, n\}$ and computes an approximate value Y such that $|Y - F_k| > F_k/10$ with probability $< 1/2$ requires at least $\Omega(n^{1-5/k})$ memory bits.*

Theorem 12 holds in a streaming scenario where items are revealed to the algorithm in an online manner and no assumptions are made on the input. We finally notice that in the same scenario there are problems for which approximation and randomization do not help at all. A prominent example is given by the computation of F_∞, the maximum frequency of any item in the stream.

Theorem 13 *[[6]] Any randomized algorithm that makes one pass over a sequence of at least $2n$ items drawn from the universe $U = \{1, 2, \ldots, n\}$ and computes an approximate value Y such that $|Y - F_\infty| \geq F_\infty/3$ with probability $< 1/2$ requires at least $\Omega(n)$ memory bits.*

8.5 SUMMARY

In this chapter we have addressed the emerging field of data stream algorithmics, providing an overview of the main results in the literature and discussing computational models, applications, lower bound techniques, and tools for designing efficient algorithms. Several important problems have been proven to be efficiently solvable

despite the strong restrictions on the data access patterns and memory requirements of the algorithms that arise in streaming scenarios. One prominent example is the computation of statistical summaries such as frequency moments, histograms, and wavelet coefficient, which are of great importance in a variety of applications including network traffic analysis and database optimization. Other widely studied problems include norm estimation, geometric problems such as clustering and facility location, and graph problems such as connectivity, matching, and distances.

From a technical point of view, we have discussed a number of important tools for designing efficient streaming algorithms, including random sampling, probabilistic counting, hashing, and linear projections. We have also addressed techniques for graph problems and we have shown that extending the streaming paradigm with a sorting primitive yields enough power for solving a variety of problems in external memory, essentially matching the results obtainable in more powerful computational models for massive data sets.

Finally, we have discussed lower bound techniques, showing that tools from the field of communication complexity can be effectively deployed for proving strong streaming lower bounds. We have discussed the role of randomization and approximation, showing that for some problems neither one of them yields enough power, unless they are used together. We have also shown that other problems are intrinsically hard in a streaming setting even using approximation and randomization, and thus cannot be solved efficiently unless we consider less restrictive computational models.

ACKNOWLEDGMENTS

We are indebted to Alberto Marchetti-Spaccamela for his support and encouragement, and to Andrew McGregor for his very thorough reading of this survey. This work has been partially supported by the Sixth Framework Programme of the EU under Contract IST-FET 001907 (“DELIS: Dynamically Evolving Large Scale Information Systems”) and by MIUR, the Italian Ministry of Education, University and Research, under Project ALGO-NEXT (“Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments”).

REFERENCES

1. Agrawal D, Metwally A, El Abbadi, A. Efficient computation of frequent and top-k elements in data stream. In: Proceedings of the 10th International Conference on Database Theory; 2005; p 398–412.
2. Abello J, Buchsbaum A, Westbrook JR. A functional approach to external graph algorithms. Algorithmica 2002; 32(3):437–458.
3. Aggarwal G, Datar M, Rajagopalan S, Ruhl M. On the streaming model augmented with a sorting primitive. In: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS’04); 2004.

4. Alon N, Gibbons P, Matias Y, Szegedy M. Tracking join and self-join sizes in limited storage. In: Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS'99); 1999; p 10–20.
5. Alon N, Matias Y, Szegedy M. The space complexity of approximating the frequency moments. In: Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC'96): ACM Press; 1996; p 20–29.
6. Alon N, Matias Y, Szegedy M. The space complexity of approximating the frequency moments. J Comput Syst Sci 1999; 58(1):137–147.
7. Babcock B, Babu S, Datar M, Motwani R, Widom J. Models and issues in data stream systems. In: Proceedings of the 21st ACM Symposium on Principles of Database Systems (PODS'02); 2002; p 1–16.
8. Babcock B, Datar M, Motwani R. Sampling from a moving window over streaming data. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02); 2002; p 633–634.
9. Bar-Yossef Z, Jayram T, Kumar R, Sivakumar D. Information statistics approach to data stream and communication complexity. In: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS'02); 2002.
10. Bar-Yossef Z, Kumar R, Sivakumar D. Reductions in streaming algorithms, with an application to counting triangles in graphs. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02); 2002; p 623–632.
11. Bar-Yossef Z, Jayram T, Kumar R, Sivakumar D, Trevisan L. Counting distinct elements in a data stream. In: Proceedings of the 6th International Workshop on Randomization and Approximation Techniques in Computer Science; 2002; p 1–10.
12. Bhuvanagiri L, Ganguly S, Kesh D, Saha C. Simpler algorithm for estimating frequency moments of data streams. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06); 2006; p 708–713.
13. Buriol L, Frahling G, Leonardi S, Marchetti-Spaccamela A, Sohler C. Counting triangles in data streams. In: Proceedings of the 25th ACM Symposium on Principles of Database Systems (PODS'06); 2006; p 253–262.
14. Chakrabarti A, Khot S, Sun X. Near-optimal lower bounds on the multi-party communication complexity of set disjointness. In: Proceedings of the IEEE Conference on Computational Complexity; 2003; p 107–117.
15. Charikar M, Chaudhuri S, Motwani R, Narasayya V. Towards estimation error guarantees for distinct values. In: Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS'00); 2000; p 268–279.
16. Charikar M, Chen K, Farach-Colton M. Finding frequent items in data streams. In: Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP'02); 2002; p 693–703.
17. Charikar M, O'Callaghan L, Panigrahy R. Better streaming algorithms for clustering problems. In: Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC'03); 2003.
18. Chaudhuri S, Motwani R, Narasayya V. Random sampling for histogram construction: How much is enough? In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 1998; p 436–447.

19. Chiang Y, Goodrich MT Grove EF Tamassia R, Vengroff DE, Vitter JS External-memory graph algorithms. In: Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'95); 1995; p 139–149.
20. Coppersmith D, Kumar R. An improved data stream algorithm for frequency moments. In: Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04); 2004; p 151–156.
21. Cormode G, Muthukrishnan S. Estimating dominance norms on multiple data streams. In: Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03); 2003; p 148–160.
22. Cormode G, Muthukrishnan S. What is hot and what is not: Tracking most frequent items dynamically. In: Proceedings of the 22nd ACM Symposium on Principles of Database Systems (PODS'03); 2003.
23. Cormode G, Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications. J Algorithms 2005; 55(1):58–75.
24. Cormode G, Muthukrishnan S. Space efficient mining of multigraph streams. In: Proceedings of the 24th ACM Symposium on Principles of Database Systems (PODS'05); 2005.
25. Demetrescu C, Finocchi I, Ribichini A. Trading off space for passes in graph streaming problems. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06); 2006; p 714–723.
26. Elkin M, Zhang J. Efficient algorithms for constructing $(1 + \epsilon, \beta)$-spanners in the distributed and streaming models. In: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04); 2004; p 160–168.
27. Feigenbaum J, Kannan S, McGregor A, Suri S, Zhang J. On graph problems in a semi-streaming model. In: Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP'04); 2004.
28. Feigenbaum J, Kannan S, McGregor A, Suri S, Zhang J. Graph distances in the streaming model: the value of space. In: Proceedings of the 16th ACM/SIAM Symposium on Discrete Algorithms (SODA'05); 2005; p 745–754.
29. Flajolet P, Martin GN. Probabilistic counting. In: Proceedings of the 24th Annual Symposium on Foundations of Computer Science; 1983; p 76–82.
30. Flajolet P, Martin GN. Probabilistic counting algorithms for database applications. J Comput Syst Sci 1985; 31(2):182–209.
31. Frahling G, Indyk P, Sohler C. Sampling in dynamic data streams and applications. In: Proceedings of the 21st ACM Symposium on Computational Geometry; 2005; p 79–88.
32. Frahling G, Sohler C. Coresets in dynamic geometric data streams. In: Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05); 2005.
33. Gibbons PB, Matias Y. New sampling-based summary statistics for improving approximate query answers. In: Proceedings of the ACM SIGMOD International Conference on Management of Data; 1998.
34. Gibbons PB, Matias Y. Synopsis data structures for massive data sets. In: External Memory Algorithms. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 50; 1999; 39–70. [Q1]
35. Gibbons PB, Matias Y, Poosala V. Fast incremental maintenance of approximate histograms. In: Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97); 1997.

36. Gilbert A, Kotidis Y, Muthukrishnan S, Strauss M. How to summarize the universe: dynamic maintenance of quantiles. In: Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02); 2002; p 454–465.
37. Gilbert AC, Guha S, Indyk P, Kotidis Y, Muthukrishnan S, Strauss M. Fast, small-space algorithms for approximate histogram maintenance. In: Proceedings of the 34th ACM Symposium on Theory of Computing (STOC'04); 2002; p 389–398.
38. Gilbert AC, Kotidis Y, Muthukrishnan S, Strauss M. Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In: Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01); 2001; p 79–88.
39. Golab L, Ozsu MT. Data stream management issues — a survey. Technical report TR CS-2003-08. School of Computer Science, University of Waterloo; 2003.
40. Guha S, Indyk P, Muthukrishnan S, Strauss M. Histogramming data streams with fast per-item processing. In: Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP'02); 2002; p 681–692.
41. Guha S, Koudas N, Shim K. Data streams and histograms. In: Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC'01); 2001; p 471–475.
42. Guha S, Mishra N, Motwani R, O'Callaghan L. Clustering data streams. In: Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS'00); 2000; p 359–366.
43. Henzinger M, Raghavan P, Rajagopalan S. Computing on data streams. In: External Memory Algorithms. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 50; 1999; 107–118. [Q1]
44. Indyk P. Stable distributions, pseudorandom generators, embeddings and data stream computation. In: Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS'00); 2000; p 189–197.
45. Indyk P. Algorithms for dynamic geometric problems over data streams. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC'04); 2004; p 373–380.
46. Indyk P, Woodruff D. Tight lower bounds for the distinct elements problem. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03); 2003.
47. Indyk P, Woodruff D. Optimal approximations of the frequency moments. In: Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05); 2005.
48. Jájá J. An Introduction to Parallel Algorithms. Addison-Wesley; 1992.
49. Jowhari H, Ghodsi M. New streaming algorithms for counting triangles in graphs. In: Proceedings of the 11th Annual International Conference on Computing and Combinatorics (COCOON'05); 2005; p 710–716.
50. Kushilevitz E, Nisan N. Communication Complexity. Cambridge University Press; 1997.
51. Manku GS, Motwani R. Approximate frequency counts over data streams. In: Proceedings 28th International Conference on Very Large Data Bases (VLDB'02); 2002; p 346–357.
52. Matias Y, Vitter JS, Wang M. Dynamic maintenance of wavelet-based histograms. In: Proceedings of 26th International Conference on Very Large Data Bases (VLDB'00); 2000.
53. McGregor A. Finding matchings in the streaming model. In: Proceedings of the 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX'05), LNCS 3624; 2005; p 170–181.

54. Misra J, Gries D. Finding repeated elements. Sci Comput Program 1982; 2:143–152.
55. Morris R. Counting large numbers of events in small registers. Commun ACM, 1978; 21(10):840–842.
56. Munro I, Paterson M. Selection and sorting with limited storage. Theor Comput Sci 12:315–323, 1980. A preliminary version appeared in IEEE FOCS'78.
57. Muthukrishnan S. Data streams: algorithms and applications. Technical report; 2003. Available at `http://athos.rutgers.edu/~muthu/stream-1-1.ps`.
58. Muthukrishnan S, Strauss M. Maintenance of multidimensional histograms. In: Proceedings of the FSTTCS; 2003; p 352–362.
59. Muthukrishnan S, Strauss M. Rangesum histograms. In: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03); 2003.
60. Muthukrishnan S, Strauss M. Approximate histogram and wavelet summaries of streaming data. Technical report, DIMACS TR 2004-52; 2004.
61. Ruhl M. Efficient algorithms for new computational models. PhD Thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology; 2003.
62. Saks M, Sun X. Space lower bounds for distance approximation in the data stream model. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC'02); 2002; p 360–369.
63. Vitter JS. Random sampling with a reservoir. ACM Trans Math Software 1995; 11(1):37–57.
64. Vitter JS. External memory algorithms and data structures: dealing with massive data. ACM Comput Surv, 2001; 33(2):209–271.

[Q1]:- Please provide the details of the publisher in References 34, 43.