Testing and Spot-Checking of Data Streams*

Joan Feigenbaum†
Computer Science Department
Yale University
New Haven, CT 06520 USA
joan.feigenbaum@yale.edu

Sampath Kannan‡
Department of Computer & Information Science
University of Pennsylvania
Philadelphia, PA 19104 USA
kannan@central.cis.upenn.edu

Martin Strauss
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932 USA
mstrauss@research.att.com

Mahesh Viswanathan§
DIMACS & Telcordia Technologies
Rutgers, The State University of New Jersey
Piscataway, NJ 08854 USA
maheshv@dimacs.rutgers.edu

March 14, 2000

Abstract

We consider the tasks of *testing* and *spot-checking* for *data streams*. These testers and spot-checkers are potentially useful in real-time or near real-time applications that process huge datasets. Crucial aspects of the computational model include the space complexity of the testers and spot-checkers (ideally much lower than the size of the input stream) and the number of passes that the tester or spot-checker must make over the input stream (ideally one, because the original stream may be too large to store for a second pass).

A sampling-tester [GGR98] for a property P samples some (but usually not all) of its input and, with high probability, outputs PASS if the input has property P and FAIL if the input is *far* from having P, for an appropriate sense of "far." A streaming-tester for a property P of one or more input streams takes as input one or more data streams and, with high probability, outputs PASS if the streams have property P and FAIL if the streams are far from having P. A sampling-tester can make its samples in any order; a streaming-tester sees the input from left to right.

We consider the *groupedness property* (a natural relaxation of the sortedness property). We also revisit the sortedness property, first considered in [EKK+98] in the context of sampling spot-checkers, and the property of detecting whether one stream is a permutation of another (either directly or via the SORTED-SUPERSET property, a technical property that is equivalent to PERMUTATION under some conditions). We show that there are properties efficiently testable by a streaming-tester but not by a sampling-tester and other (promise) problems for which the reverse is true.

*This paper is an expanded version of the authors' 2000 ACM-SIAM Symp. on Discrete Algorithms extended abstract with the same title.

†Work done while the author was a member of the Information Sciences Research Center of AT&T Labs in Florham Park, NJ.

‡Supported by grants NSF CCR98-20885 and ARO DAAH04-95-1-0092. Part of this work was done at AT&T Labs in Florham Park, NJ.

§This work was done at the University of Pennsylvania, supported by grant ONR N00014-97-1-0505, MURI.

1 Introduction

Massive data sets are increasingly important in a wide range of applications, including observational sciences, product marketing, and monitoring and operations of large systems. In network operations, raw data typically arrive in *streams*, and decisions must be made by algorithms that make one pass over each stream, throw much of the raw data away, and produce "synopses" or "sketches" for further processing. The enormous scale, distributed nature, and one-pass processing requirement on the data sets of interest must be addressed with new algorithmic techniques.

Two programming paradigms for massive data sets are *sampling* and *streaming*. Rather than take time even to read a massive data set, a sampling algorithm extracts a small random sample and computes on it. By contrast, a streaming algorithm takes time to read all the input, but little more time and little total space. Input to a streaming algorithm is a sequence of items; the streaming algorithm is given the items in order, lacks space to record more than a small amount of the input, and is required to perform its per-item processing quickly in order to keep up with the unbuffered input. UNIX pipelines and applications monitoring high-bandwidth network connections are practical examples of streaming algorithms. Formulations of models for streaming algorithms can be found in, *e.g.*, [HRR98, FKSV99a].

Recently *spot-checkers* [EKK$^+$98] and, earlier, *property testers* [GGR98] have been proposed. These are sampling programs that perform variants of *program checking* [BK95, BLR93]. A property tester T_P for a property P gets as input a data set x purported to have P. The program T_P samples a small portion of x, runs quickly, and, with high probability, outputs PASS if x has P and outputs FAIL if x is *far* from any x' with P (for an appropriate sense of "far"). A spot-checker C_f for a function f gets (x, y) purported to satisfy $f(x) = y$; C_f samples a small portion of its input and outputs PASS or FAIL depending on whether the pair (x, y) indeed satisfies $f(x) = y$ or is far from any satisfying pair. A property tester might be run before a primary program in order to validate input for the latter; a spot-checker might be run after a primary program to insure that the latter did not fail egregiously. The small runtime of checkers/testers is especially important when the primary program uses little time—for example, if the primary program is itself a sampling program. This is less important if the primary program takes time to read all its input.

In this paper, we introduce *streaming-property-testers* and *streaming-spot-checkers*. A streaming-property-tester T_P for a property P takes as input a read-once data stream x, uses little space and little time per item, and, with high probability, outputs PASS if x has P and outputs FAIL if x is far from any x' with P. A streaming-spot-checker for a function f has a similar definition. A streaming-property-tester may be used to validate input for a primary pipeline of streaming algorithms; a streaming spot-checker can insure that the pipeline did not fail egregiously. (We assume that invalid input is rare. Thus, we are content with a property tester that catches invalid input eventually, even if the primary program performs work before the bad input is caught.) A collection of primary programs might be written as a pipeline (rather than as a sequence of programs, each idling until the previous program terminates) when (1) it is acceptable or necessary to expend enough time to read the input, and (2) it is not acceptable to store intermediate results from individual primary programs. In a context suited for pipelines, the guarantees given about (sampling) spot-checkers and property testers are inappropriate because (1) the significantly sublinear time guarantees of the sampling-checkers and sampling-testers are useless because the checkers/testers can be run at the same time as the primary program anyway, and, (2) a sampling-checker/tester may require storage of the input or output (or even storage of the input until the output is finished, for example, to sample from the input using a distribution determined by the output), whereas a streaming-checker or streaming-tester is guaranteed not to increase significantly the space requirements of the primary program. Furthermore, for some properties (such as GROUPEDNESS discussed below), it is easier and more natural to design streaming-testers, which are sensitive to all the input, rather than sampling-testers, which are sensitive to a small sample only. Thus, designing a streaming-tester is sometimes easier than designing and modifying a sampling-tester, and, even though some sampling-checkers and sampling-testers can be modified to work well in a streaming context, it is important to analyze them in a streaming context in order to establish appropriate performance guarantees.

We consider whether every property that can be efficiently tested by a sampling-tester can also be efficiently tested by a streaming-checker, and *vice versa.* The notions of efficiency for sampling-testers and streaming-testers are not directly comparable, but there are still interesting comparisons to make. There are useful, non-trivial sampling-checkers and sampling-testers (for two different problems) that, respectively, run in time $O(\log n)$ [EKK$^+$98] and $O(\sqrt{n}\log(n))$ [here]. Both are the best possible up to a log factor. Thus, even though a runtime of $\tilde{O}(\sqrt{n})$ represents substantial savings over any algorithm that reads all of its input, it is not the best one can achieve for spot-checkers generally. Because there are functions/properties checkable in $\log^{O(1)} n$ time, we reserve the term "efficiently checkable/testable" for that. As for streaming-checkers and streaming-testers, we call a function/property "efficiently stream-checkable" or "efficiently stream-testable" if it uses $O(\log n)$ space and at most time $\log^{O(1)} n$ per item. We show that there are properties efficiently testable by sampling-testers but not by streaming-testers and other properties for which the reverse is true. We do this by considering the SORTEDNESS property, the PERMUTATION property that one half of the input is a permutation of the other half, and the GROUPEDNESS property. (A sequence $\sigma_1, \ldots, \sigma_n$ is called *grouped* if $\sigma_i = \sigma_j$ and $i < k < j$ imply $\sigma_i = \sigma_k = \sigma_j$, *i.e.,* for each type T, all occurrences of T are in a single (contiguous) run.)

1.1 Summary

Our main technical contribution is showing that a sampling-tester can test whether a sequence of length n is grouped, making only $O(\sqrt{n})$ samples using $O(\sqrt{n}\log(n))$ time and $O(\sqrt{n})$ space. (The proof is given in Section 3.) This, along with related and complementary results proved here or elsewhere, is important because it (1) gives a non-trivial property-tester for a property—groupedness—that is important for algorithms for massive data sets, (2) demonstrates that the guarantees given by the paradigm of sampling-checkers and sampling-testers are inappropriate for streaming algorithms, while the paradigm of streaming-checking and testing that we introduce gives more useful guarantees, and (3) demonstrates that there are properties efficiently testable by sampling-testers but not by streaming-testers and other properties for which the reverse holds.

1.2 Relevance to Network Data

1.2.1 Streams

Data streams are a pervasive and important fact of life in communication-network operations, monitoring, and research. Basic Internet routing is itself a ubiquitous example of a streaming algorithm: A packet arrives at a router and is sent immediately on to the next hop in the path from its source to its destination; the router cannot wait and make next-hop decisions based on related packets that may arrive later, even if such decisions would lead to more efficient traffic management overall. The sampling approach to efficiency of massive-data-set processing is not relevant, because the the data set is never available in its entirety to be sampled.

Data packets form the primary streams handled by an IP network, but they are not the only streams of interest. Network-monitoring tools often produce output that is sufficiently voluminous to warrant processing as a stream. For example, Cisco routers can now be instrumented with the NetFlow feature [CN98]. As packets travel through the router, the NetFlow software produces summary statistics on each *flow*.[1] The fields of a flow record include source IP address, destination IP address, total number of packets in the flow, and total number of bytes of data in the flow. Algorithms for space-efficient computation on streams of NetFlow records can play important roles in a network-monitoring toolkit. For example, in [FKSV99a], we give a one-pass, space-efficient algorithm for approximating the L^1 difference between two streams $\langle (x, f_t(x)) \rangle$, $t = 1, 2$, where x is a source-destination pair, and $f_t(x)$ is the total number of bytes sent from the source to the destination during a time interval t; this difference is one good way to approximate the extent to which traffic patterns changed between time intervals. The key point for purposes of this discussion is that NetFlow records, although they are *summary* data, have to be processed as streams: Each WorldNet gateway router now generates more that 10Gb of NetFlow records each day.

Finally, note that networking researchers receive data streams from which they build, update, and analyze abstract models of network behavior and user behavior. The best known example of such a data stream comes from telephony: "Call-detail" records, the fields of which include originating number, terminating number, time and date of call, duration of call, and tariff- and calling-plan data for billing, are used by AT&T Labs researchers to build and analyze "call graphs"; these are massive, sparse multigraphs in which each node represents a phone number and each arc $A \rightarrow B$ represents a call from A to B, and their structural properties are both challenging to compute and useful for marketing and infrastructure planning [ABW98, ACL99, F99]. Call-detail data feeds must be processed as streams because of their size: The network handles between 200M and 300M calls a day and generates over 7GB of call-detail data.

1.2.2 Stream Checking and Testing

Because network-data collection is currently dynamic, distributed, and unreliable in nature, "sanity checks" should be performed on streams both before and after they undergo processing of various sorts. By "distributed," we mean that collection points are located at several (sometimes many) points in the network and may produce streams that are supposed to be aggregated somewhere in the network, *e.g.*, for warehousing, billing, or research purposes; for example, NetFlow records can be produced by any router in a network, web-service statistics and web-link structure can be gathered by many concurrently-running crawlers, and call-detail records are produced by many telephone-network switches. Unreliability of network-generated data arises for at least two reasons. One is that network operators will typically require that monitoring data such as NetFlow records be sent from collection points to usage points only when customer demand is low and, even during these non-peak intervals, will assign lower priority to monitoring data than to customer traffic; thus, collection points must be prepared to deal with scheduling uncertainty, delays, and errors in monitoring data streams. Another major cause of unreliability or inconsistency is third-party data sources; telephone bills, for example, are often produced by numerous different third-party billing companies, and telephone service providers often use billing records to build call graphs.

[1]Roughly speaking, a "flow" is a semantically coherent sequence of packets sent by the source and reassembled and interpreted at the destination. Any precise definition of "flow" would have to depend on the application(s) that the source and destination processes were using to produce and interpret the packets. From the router's point of view, a flow is just a set of packets with the same source and destination IP addresses whose arrival times at the routers are close enough, for a tunable definition of "close."

1.2.3 Groupedness

We study groupedness primarily because, like sortedness, it is a fundamental property that one may want a massive data set to have and because it illustrates that some properties are efficiently testable by sampling algorithms but not streaming algorithms and *vice versa*. However, it is also worth noting that groupedness is a particularly desirable property for network data that must be processed in a streaming fashion, because many common uses of network-generated massive data streams require computing maxima. For example, suppose, on a specific day, a network router routes a_i packets to destination i and needs to report the number of packets routed to the most popular destination. This computation is trivial if the packets are guaranteed to be grouped by destination, but, if they are not grouped, even approximating the desired value with substantial probability requires space essentially sufficient to store the entire data stream [AMS96]. Similarly, grouped data streams facilitate reporting most frequently accessed web pages, most frequently called 800 numbers, most frequently used services, *etc.* Finally, we remark that grouping algorithms are a key enabler of recent improvements in the I/O complexity of semi-external graph algorithms [ABW98]. Semi-external computation of basic graph structures, *e.g.*, connected components, minimum spanning trees, and matchings, is necessary when dealing with call graphs, web graphs, and other massive, sparse graphs arising from communications networks.

2 Definitions and Computational Models

Our model is closely related to that of Henzinger, Raghavan, and Rajagopalan [HRR98], but variants of our model also incorporate features of spot-checking [EKK+98] and property-testing [GGR98].

2.1 The Streaming Model

As in [FKSV99a, HRR98], a *data stream* is a sequence of data items $\sigma_1, \sigma_2, \ldots, \sigma_n$ that, on each *pass* through the stream, are read once in increasing order of their indices. In this computational model, we assume that the input is one or more data streams. Again, as in [HRR98], we focus on two resources—the *workspace* required and the maximum *number of passes* through any input stream. A point of departure from [HRR98] is that we implicitly assume that the time complexity of each pass is at most linear in input size. When we deviate from this assumption, we will explicitly state our time bounds. Often only constant time is used between reading the i'th and $(i+1)$'st items in a stream.

We now describe the spot-checking and property-testing framework, because we would like to situate our model within this framework.

Spot-checking is a variant of program checking that ensures in sublinear time that the output of a program is *approximately correct*. Assume that D is a distance function on the space of pairs the first element of which is in the domain of f and the second of which is in the range of f.

Definition 1 An **ϵ-spot-checker** for function f under distance function D is itself a probabilistic algorithm C that, on input $\langle x, y\rangle$, has the following properties.

1. If $f(x) = y$, then C outputs *PASS* with high probability.
2. If $D(\langle x, y\rangle, \langle x', f(x')\rangle) \geq \epsilon$, for all $\langle x', f(x')\rangle$, then C outputs *FAIL* with high probability.
3. The checker C runs in time $o(|x| + |y|)$.

∎

A property-tester is similar in spirit to a spot-checker. Assume that D is a distance function on the space of inputs to a property P.

Definition 2 An **ϵ-property-tester** for property P under distance function D is itself a probabilistic algorithm T that, on input x, has the following properties.

1. If $x \in P$, then T outputs *PASS* with high probability.
2. If $D(x, x') \geq \epsilon$, for all $x' \in P$, then T outputs *FAIL* with high probability.
3. The tester T runs in time $o(|x|)$.

∎

Remark 1 *A spot-checker for the function $f : X \to Y$ is a property-tester for f viewed as a property on $X \times Y$ with the same distance function, D.*

Remark 2

In [GGR98], a property-tester is always defined with respect to the Hamming distance.

The non-triviality requirement that spot-checkers run in sublinear time insures that they don't merely duplicate the computation of the function f being checked. Since spot-checkers run in sublinear time for each choice of random bits, the checker sees less than all the input and output of f. For our analysis, it is convenient to force a checker to access f's input and output via an explicit query operation, so that we can count the number of queries made and thereby define the *query complexity* or, synonymously, the *sample complexity* of problems. As in [GGR98], we are interested both in the query complexity and the runtime, but, generally, we give no explicit bound on the workspace. (Note that the workspace used is bounded by the runtime, which is often much less than linear.)

Our goal here is to define a model of streaming checking that allows spot-checking and property-testing for inputs that are streamed. While existing spot-checkers are bounded in their runtime, streaming spot-checkers will be bounded in their workspace. Because runtime and one-way workspace are resources with different characteristics, there are functions that have [EKK$^+$98]-spot-checkers but not streaming spot-checkers, and *vice versa*. (We give examples in section 3.) The same remark applies also to streaming property-testers, *mutatis mutandis*.

We can now define the idea of streaming-property-testing, a variant of property-testing in which the input is a data stream. Assume as before that there is a distance function D defined on the space of inputs.

Definition 3 An **ϵ-streaming-property-tester** for a property P under distance D is a probabilistic algorithm that takes an input stream S and behaves as follows:

1. If S has property P, then the checker outputs *PASS* with high probability.
2. If $D(S, S') \geq \epsilon$ for any stream S' that has property P, then the tester outputs *FAIL* with high probability.
3. The space complexity of the tester is $o(|S|)$.

■

Similarly we can define the notion of a *streaming spot-checker*. Let f be a transducer that takes in a stream x and outputs a stream y. Assume that D is a distance on input-output pairs $\langle x, y\rangle$.

Definition 4 An **ϵ-streaming-spot-checker** for function f under distance function D is itself a probabilistic algorithm C that, on input $\langle x, y\rangle$, has the following properties.

1. If $f(x) = y$, then C outputs *PASS* with high probability.
2. If $D(\langle x, y\rangle, \langle x', f(x')\rangle) \geq \epsilon$, for all $\langle x', f(x')\rangle$, then C outputs *FAIL* with high probability.
3. The space complexity of the checker is $o(|x| + |y|)$.

■

Thus, a streaming spot-checker takes as input two streams. One can consider a model in which, at each computation step, the streaming algorithm specifies which of the two or more input stream pointers to advance; alternatively, an adversary may advance the streams out of the control of the algorithm. We will specify the model further as we consider specific problems.

Finally, we can generalize streaming-spot-checkers and streaming-property-testers to multi-pass algorithms in the natural way.

In [HRR98], the authors propose a study of the streaming model under several dichotomies — one pass versus multi-pass, deterministic versus randomized, and exact versus approximate. In this paper, our focus on property testing and spot-checking leads naturally to the study of randomized and approximate algorithms. We will separate functions checkable by space-bounded streaming spot-checkers (that can see all the input) and functions checkable by traditional, random-access, time-bounded spot-checkers (that can see only a portion of the input but can see it adaptively).

3 Testers for Grouped Sequences

In this section, we give upper and lower bounds for the sampling and streaming complexity of testing and checking groupedness. These results, together with known results ([EKK$^+$98] and elsewhere) about the related sortedness problem, highlight differences between the streaming and sampling models.

Definition 5 A sequence of elements $\sigma = \sigma_1\sigma_2\cdots$ is said to be **grouped** if, for all $i < k < j$, if $\sigma_i = \sigma_j$ then $\sigma_i = \sigma_k = \sigma_j$. ■

In other words, a sequence is grouped if all the elements of each kind occur consecutively in the sequence.

Definition 6 The distance of a sequence σ from a grouped sequence is the minimum fraction of elements of σ that need to be moved in order to get a grouped sequence. ■

This notion of distance is comparable to the one used in [EKK$^+$98] for sorting. There, a sequence σ of length n is at distance ϵ from being *sorted* if the longest sorted subsequence has length $(1-\epsilon)n$. Observe that the length of the longest sorted subsequence is $(1-\epsilon)n$ if and only if the minimum fraction of elements that need to be *changed* in order to get a sorted sequence is ϵ (*i.e.*, σ is at Hamming distance ϵn from sorted). A transformation moving the minimal number of elements is called *optimal*.

Figure 1: A sampling-tester for groupedness.

Input: Sequence σ of length n

Initialize:
Pick $O(k\sqrt{n})$ indices uniformly at random from the set $\{0,\ldots,n-1\}$, where $k = \frac{1}{\sqrt{2\epsilon}} + 1$.
Let the sorted sequence of these indices be $i_1, i_2, \ldots, i_{O(\sqrt{n})}$.
Pick an index uniformly at random between i_j and i_{j+1} for each j.
(Choose all the indexes, then make all the samples.)

If there is a violating triple in the sample, *i.e.*, there are indices $i < j < \ell$ in the chosen sample such that $\sigma_i = \sigma_\ell$ and $\sigma_i \neq \sigma_j$, then
return FAIL
else return PASS

3.1 A Sampling-Tester

An ϵ-sampling-property-tester for groupedness is given in Figure 1.

Theorem 7 *Let σ be a sequence of n elements from $[m] = \{0,\ldots,m-1\}$. There is a single pass ϵ-sampling-property-tester for groupedness of σ. It uses $O(\sqrt{n})$ space, makes $O(\sqrt{n})$ samples, and runs in $O(\sqrt{n}\log n)$ time (assuming a single item from $[m]$ can be processed in constant time and space).*

The proof makes use of the following definition and lemma.

Definition 8 A pair of indices (i,j), where $i < j$, is said to be **bad** for a sequence σ, if $\sigma_i = \sigma_j$ and, for at least half the positions k strictly between i and j, $\sigma_k \neq \sigma_i$. ∎

Lemma 9 *Let σ be a sequence of length n that is at distance ϵ from being grouped,* i.e., *at least ϵn elements of σ need to be moved in order to get a grouped sequence. Then there are at least $\epsilon n/2$ bad pairs in σ, no two of which share even one endpoint.*

Proof.
We'll start with a maximal endpoint-disjoint collection C of bad pairs. From C, we construct a correct transformation τ with a number of moves bounded by at most $2|C|$. It follows that an optimal transformation requires at most $2|C|$ moves.

Let C be a maximal collection of bad pairs such that no two bad pairs share even one endpoint. We'll say that an element of σ is *covered* if it is in a pair in C. We will also abuse notation and say that elements of σ themselves, rather than pairs, are elements of C. We let $|C|$ denote the number of bad pairs, so that the number of covered elements is $2|C|$.

Let U consist of blocks of uncovered elements of like type. (A block of type A is terminated either by an element of type other than A or by a covered element of type A.)

Consider the following transformation, which consists of two phases. First, we group the blocks of uncovered elements in a particular way, then we move all covered elements to the correct place in the resulting sequence.

Let A be the type of the leftmost block in U. We move all non-A's out from between the leftmost and rightmost type-A block in U (if there is more than one such block), to the appropriate place (specified later). Next, let B be the type of the leftmost block in U to the right of the rightmost type-A block in U; move all non-B's out from between the leftmost and rightmost type-B blocks to the right of the A blocks. Repeat this procedure, from left to right, for all dynamically remaining blocks in U. At this point, we know where to move the elements moved out from between blocks—either to the unmoved elements of the correct type, if such elements exist, or to an end of the sequence, otherwise. Finally, move all covered elements to the correct place. By construction, this transformation is correct.

We assign to each move a unique covered element. If an element moves in the second phase, because it is covered, we assign it to itself. Now, consider two blocks B_1 and B_2 in U of like type, A, such that there is no other type-A block between B_1 and B_2 and such that all non-A's between the blocks are moved out. Since the rightmost A of the left block and the leftmost A of the right block is not a bad pair, at least half of the elements between these two block-end A's are also A's, and must be covered, non-moving A's. We assign the moving non-A's between the block-end A's to the more-plentiful covered A's between the block-end A's.

Note that the moving covered A's are assigned themselves (elements of the same type) and the non-moving covered A's are assigned only elements of type other than A, so the assignment to covered A's is injective.

Finally, note that all moving elements are assigned to some covered element. This is because an element moves either because it is between two blocks of like types or because the element itself is covered. ∎

Proof. [of Theorem 7] Now if the sequence is grouped, then our tester PASSes the sequence. Suppose the sequence σ, is ϵ-far from being grouped. From Lemma 9, we know that there is an endpoint-disjoint collection of at least $O(\epsilon n)$ bad pairs. Now the probability that a particular bad pair from the collection is picked, in the first step, is at least $(k\sqrt{n}/n)^2$, where $k = \frac{1}{\sqrt{2\epsilon}}+1$. The probability of picking a particular bad pair increases slightly, conditioned on knowing that we failed to pick some other bad pairs. Thus the probability that no bad pairs are picked in the first step is at most $[1-(k\sqrt{n}/n)^2]^{\epsilon n} = [1-(k^2/n)]^{\epsilon n}$. Now $[1-(k^2/n)]^{\epsilon n} < 1 - \epsilon k^2 + \epsilon^2 k^4 = 1-(1/2)+(1/4) = (3/4)$. Hence the probability that a bad pair is picked in the first step is at least $1/4$. Now since for a bad pair at least half the elements in between are different, the probability that we get a witnessing triple in our sample is at least $(1/4)(1/2) = 1/8$. Thus with a probability $1/8$ our tester detects that the sequence is not grouped. Note that this success probability can be boosted up to any $1-\delta$ by conducting $\log(1/\delta)$ many such experiments simultaneously and rejecting if any of those experiments FAIL the sequence.

The algorithm needs $O(\sqrt{n})$ storage space to store the sample indices. After making samples, the algorithm can then create $O(\sqrt{n})$ triples corresponding to the sample; each triple has the type of the element, its index in the original sequence, and the index of the next element in the sample. We impose an arbitrary ordering on the types and then sort this sequence of triples, first based on type and then on index. Then, if, while passing through the sorted sample, we come across a triple (a, i, j) such that the element immediately following it, in this sorted sequence, is (a, k, l), where $k \neq j$, then we have discovered a violating triple, namely (i, j, k). Conversely, if there is a violating triple (i, j, k) then, assuming without loss of generality that i is maximal for these values of j and k, we will find a violating triple this way. All this takes only $O(\sqrt{n}\log n)$ time and $O(\sqrt{n})$ space. ∎

The above tester is, in fact, optimal in terms of the number of samples needed.

Figure 2: A streaming-tester for groupedness.

Input: Sequence σ of length n

Initialize:

Pick $O(k)$ indices uniformly at random from the set $\{0, \ldots, n\}$, where $k = O\left(\frac{1}{\epsilon}\right)$.
Let the sorted sequence of these indices be $i_1, i_2, \ldots, i_k$.

If there is a violating triple whose first component is in the index set,
i.e., if there are indices $i < j < \ell$, with i in the
chosen index set such that $\sigma_i = \sigma_\ell$ and $\sigma_i \neq \sigma_j$, then
return FAIL
else return PASS

Theorem 10 *Any sampling-tester for groupedness makes $\Omega(\sqrt{n})$ samples on input of size n.*

Proof. Suppose, by contradiction, that there is a sampling-tester for groupedness that makes $k = o(\sqrt{n})$ non-adaptive queries. Consider a universe of $\{1, \ldots, n\}$. If the tester gets any permutation of the universe, then it will see k different elements, and must answer PASS because the sequence is grouped. Now, consider a sequence that contains two each of $n/2$ symbols, chosen uniformly at random from $\{1, \ldots, n\}$ without replacement, permuted in a random order. The tester will, with high probability, see a subset of k different elements, and so will output PASS. With high probability, however, such sequences are far from grouped. In particular, for each tester T, there exists a far-from-grouped sequence σ_T, such that T incorrectly PASSes σ_T.

Now consider an adaptive tester. Such a tester will also output PASS when its set of samples has all distinct elements. On the other hand, suppose the tester is given a non-grouped sequence generated as in the previous paragraph. By induction on $k' = 1, 2, \ldots, k$, after seeing k' samples, the tester would not have seen a match and could therefore have made its $k + 1$'st sample non-adaptively. It follows that the adaptive tester is equivalent to a non-adaptive tester and will fail on some far-from-grouped sequence. ∎

Thus the sampling complexity is $\Omega(\sqrt{n})$, ignoring dependence on ϵ and δ.

3.2 A Streaming-Tester

In Section 3.1, we showed there are no efficient sampling-testers for groupedness. On the other hand, there *are* efficient streaming-testers for groupedness, *i.e.*, streaming-testers that make one pass, use logarithmic space, and polylogarithmic time per item.

Theorem 11 *There exists a single-pass ϵ-streaming-property-tester for groupedness. It uses $O(1)$ space (ignoring dependence on ϵ and on the number m of types).*

The tester is give in Figure 2. In order to prove this theorem we need the following definition and lemma.

Definition 12 An index i is said to be **streaming-bad** for a sequence σ, if there exists j, ℓ, $i < j < \ell$, such that $\sigma_i \neq \sigma_j$ but $\sigma_i = \sigma_\ell$. ∎

In the context of this proof, we simply say "bad."

Lemma 13 *Let σ be a sequence that is ϵ-far from being grouped,* i.e., *at least ϵn elements of σ need to be moved in order to get a grouped sequence. Then there are at least ϵn bad indices for σ.*

Proof. If we remove the bad indices from a sequence, the resulting sequence is grouped. ∎

Proof.[of Theorem 11] Clearly the algorithm PASSes grouped sequences. Suppose an input sequence is not grouped.

Because the number of bad indices has density ϵ, we are likely to find one by sampling $O(1/\epsilon)$ indices. It is straightforward to test each index sampled to see whether or not it is bad, using just constant space per index (plus $O(\log m)$ space per index to store the name of the index). ∎

3.3 Checkers

Above we considered testers for groupedness, which take as input a single sequence, purportedly grouped. By contrast, we now consider checkers for groupedness, which take as input two sequences (an *input* sequence and an *output* sequence), where the output is purported to be a grouped permutation of the input. Thus a checker for groupedness needs to test the output for groupedness *and* test whether the output is a permutation of the input.

For a streaming-checker, we regard the input and output to be concatenated into a single sequence. Alternatively, we may consider the checker to work on two streams and that, at each computation step, an adversary chooses which of the stream pointers to advance.

Definition 14 A sequence of $2n$ symbols from $[m]$ is said to be a **PERMUTATION** if the final n symbols represent the same subset of $[m]$ as the initial n symbols.

We use the Hamming distance for checkers of PERMUTATION. ∎

Proposition 15 *For any $\epsilon > 0$, there is an ϵ-streaming-tester for* PERMUTATION *that uses $O(\log(n))$ space.*

Proof. The algorithm is as follows:

Let F be a finite field with approximately n^2 elements. Pick $x \in F$ uniformly at random. If type i occurs a_i times in the first n positions and b_i times in the second n positions, compute the polynomial $P = \sum(a_i - b_i)x^i$. If $P = 0$ output PASS; else output FAIL.

Note that the algorithm clearly PASSes permutations. If the first and second half are not permutations, then P is a polynomial of degree at most n, so has at most n roots. The probability that we output PASS is the small probability n/n^2 that we pick a root x.

Finally, note that P can be computed in the given space by accumulating $\pm x^i$ whenever an item of type i is seen in the stream. ∎

On the other hand, as suggested in [EKK+98], PERMUTATION is difficult for sampling-testers.

Proposition 16 *For $m \geq 2n$ and $\epsilon = 1/2$, any ϵ-sampling-tester for* PERMUTATION *requires $\Omega(\sqrt{n})$ samples.*

Proof. Let each of the first and last n elements be a random subset of n distinct items from $[m]$, for $m \geq 2n$. If a checker samples $k < \sqrt{n}$ items it is likely to get k different values, whether the first and last n elements are permutations, disjoint, or somewhere in between. ∎

Finally, consider the promise problem SORTED-SUPERSET:

Definition 17 Fix a universe $[m]$. We consider sequences of the form $\alpha 0 \beta$, where α and β represent subsets of $[m]$ and α is sorted (the *promise*). An ϵ-tester for the promise problem **SORTED-SUPERSET** outputs *PASS* for sequences of this type such that $\alpha \supseteq \beta$, outputs *FAIL* for sequences of this type such that at least an ϵ fraction of β are not elements of α, and may answer arbitrarily if the sequence is of the wrong type or if the fraction of elements of β outside α is positive but less than ϵ. (Note that α and β are subsets of $[m]$, not multisets.) ∎

In [EKK$^+$98], the authors point out that a checker for sortedness under a natural notion of distance must test that the output is sorted *and* that the most of the input are elements of the output, given a promise that the output is sorted. This is essentially the SORTED-SUPERSET promise problem (for an adaptive sampling-tester, it doesn't matter whether or not α precedes β in the input representation). They noted that the SORTED-SUPERSET promise problem can be done by a sampling-tester whereas superset alone, without the sortedness promise, may be difficult. The authors show that SORTED-SUPERSET has a tester that makes $O(\log(n))$ (adaptive) samples (it performs binary search). On the other hand, for any $\epsilon < 1$, any one-pass ϵ-streaming-tester for SORTED-SUPERSET requires $\Omega(n)$ space [KN97] (consider the case $|\beta| = 1$, and note that it is crucial that α be scanned before β).

Table 1: Summary of groupedness and related testing results. Note that testing PERMUTATION or SORTED-SUPERSET is part of spot-checking sortedness or groupedness.

	GROUPEDNESS	SORTEDNESS	PERMUTATION	SORTED-SUPERSET
# samples by a sampling tester	$\Theta(\sqrt{n})$	$O(\log n)$ [EKK$^+$98]	$\Omega(\sqrt{n})$	$O(\log n)$ [EKK$^+$98]
streaming space	$O(\log n)$	trivial	$O(\log n)$	$\Omega(n)$ [KN97]

3.4 Summary

Table 1 presents a summary of the results discussed and proven in this section. We consider the case $m = \Theta(n)$ and constant ϵ. Note that two different resources are being measured—the number of samples in one case and the workspace in the other case. Nevertheless, one can consider a sampling-tester to be "efficient" if it makes $O(\log n)$ samples and a streaming-tester to be efficient if it uses $O(\log n)$ space.[2]

[2]As further justification to compare apples and oranges, note that it is often the case that the number of samples or space needed is simply the number of parallel repetitions of unit-cost work in the appropriate model.

If one defines efficiency this way, then streaming-testers/checkers and sampling-testers/checkers are incomparable. Table 1 also contains information about the complexity of groupedness and sortedness *checkers*, which must also test something like PERMUTATION or SORTED-SUPERSET. Note that an efficient sampling checker for sortedness can test SORTED-SUPERSET but not PERMUTATION, whereas the reverse is true for efficient one-pass streaming-checkers. Thus a sampling-checker and a streaming-checker solve markedly different problems to ensure that the output of the program being checked is the same as the input.

4 Acknowledgements

We would like to thank Michael Mitzenmacher and an anonymous referee for their comments on earlier drafts of this paper.

References

[ABW98] J. Abello, A. Buchsbaum, and J. Westbrook. A Functional Approach to External Graph Algorithms In *Proc. of 6'th European Symp. on Algorithms*, LNCS Vol. 1461, Springer, Berlin, 1998, pages 332-434. Expanded version submitted for journal publication.

[ACL99] W. Aiello, F. Chung Graham, and L. Lu. A Random Graph Model for Massive Graphs. `http://math.ucsd.edu/~fan/random.ps`

[AMS96] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of 28'th ACM Symposium on Theory of Computing*, pages 20–29, 1996. Expanded version to appear in *Journal of Computing and System Sciences.*

[BK95] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, January 1995. Earlier version in *Proc. of the 21'st ACM Symposium on Theory of Computing*, pages 86–97, 1989.

[BLR93] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47(3):549–595, December 1993. Earlier version in *Proc. of the 22'nd ACM Symposium on Theory of Computing*, pages 73–83, 1990.

[CN98] Cisco NetFlow, 1998. `http://www.cisco.com/warp/public/732/netflow/`.

[EKK+98] F. Ergün, S. Kannan, S. R. Kumar, R. Rubinfeld, and M. Viswanathan. Spot-checkers. In *Proc. of the 30'th ACM Symposium on Theory of Computing*, pages 259–268, 1998.

[F99] J. Feigenbaum. Massive Graphs: Algorithms, Applications, and Open Problems. Plenary talk, American Mathematical Society Winter Meeting, January 1999.

[FKSV99a] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An Approximate L^1-Difference Algorithm for Massive Data Streams. *Proc. of the 40'th IEEE Symposium on the Foundations of Computer Science*, pages 501–511, 1999.

[FKSV99b] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. Testing and Spot-Checking of Data Streams. In *Proc. of the 11'th ACM-SIAM Symp. on Discrete Algorithms*, to appear, 2000.

[GGR98] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, 45(4):653–750, July 1998.

[HRR98] M. Rauch Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report 1998-011, Digital Equipment Corporation Systems Research Center, May 1998.

[KN97] E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 1997.