Space-Efficient Straggler Identification in Round-Trip Data Streams Via Newton's Identities and Invertible Bloom Filters

David Eppstein and Michael T. Goodrich

Dept. of Computer Science, Univ. of California, Irvine, 92697

Abstract. We study the *straggler identification* problem, in which an algorithm must determine the identities of the remaining members of a set after it has had a large number of insertion and deletion operations performed on it, and now has relatively few remaining members.

1 Introduction

Imagine a security guard, who we'll call Bob, working at a large office building. Every day, Bob comes to work before anyone else, unlocks the front doors, and then staffs the front desk. After unlocking the building, Bob's job is to check in each of a set of n workers when he or she enters the building and check each worker out again when he or she leaves. Most workers leave the building by 6pm, when Bob's shift ends. But, at the end of Bob's shift, there may be a small number, at most $d << n$, of *stragglers*, who linger in the building working overtime. Before Bob can leave for home, he must tell the night guard the ID numbers of all the stragglers. The challenge is that Bob has only a small clipboard of size $o(n)$ to use as a "scratch space" for recording information as workers come and go. That is, Bob does not have enough room on his clipboard to write down all ID numbers of the workers as they arrive and check them off again as they leave. Of course, he also has to deal with the fact that some of the n workers may not come to work at all on any given day. The question we address in this paper is, "How can Bob, the security guard, check workers in and out so as to identify all d stragglers at the end of his shift, using a scratch space of size only $o(n)$?"

Formally, suppose we are given a universe $U = \{x_1, x_2, \ldots, x_n\}$ of unique identifiers, each representable with $O(\log n)$ bits. Given an upper bound parameter $d << n$, the *straggler identification problem* is to design a data structure that uses only $o(n)$ bits and efficiently supports the following operations on an initially-empty subset S of U:

- **Insert** x_i: Add the identifier x_i to S.
- **Delete** x_i: Remove the identifier x_i from S.
- **ListStragglers**: Test whether $|S| \le d$, and if so, list all the elements of S.

F. Dehne, J.-R. Sack, and N. Zeh (Eds.): WADS 2007, LNCS 4619, pp. 637–648, 2007.

We assume, without loss of generality, that d is small enough so that $d \log(n/d)$ is $o(n)$, since we need $\Omega(d \log(n/d))$ bits just to produce the answer to an **ListStragglers** query, and if d is close to n we might as well just store all the elements of S explicitly. That is, we are interested in an implicit representation of S, which can be used to list the contents of S when $|S| \leq d$, but makes no such guarantees when $|S| > d$.

In addition to our motivating example of Bob, the security guard (which also applies to other in-and-out physical environments, like amusement parks), the straggler identification problem has the following potential applications:

- In a high bandwidth multicast data stream, a server sends packets to many different clients, which send acknowledgments back to the server identifying each packet that was successfully received. The server then needs to identify and re-send the packets to clients that did not successfully receive them. This round-trip data stream application is an instance of the straggler identification problem, since we expect most of the packets to be sent successfully and we would like to minimize the space needed per client at the server for unacknowledged packet identification.
- In heterogeneous Grid computations, a supervisor sends independent tasks out to Grid participants, who, under normal conditions, perform these tasks and return the results to the supervisor. There may be a few participants, however, who crash, are disconnected from the network, or otherwise fail to perform their tasks. The supervisor would like to identity the tasks without responses, so that they can be sent to other participants for completion.

Our Results. In this paper, we study the straggler identification problem, showing that it can be solved with small space and fast update times. We provide a deterministic solution, which uses $O(d \log n)$ bits to represent the dynamic set S of $O(\log n)$-bit identifiers. Our solution is based on a novel application of Newton's identities and allows for insertions and deletions to be done in $O(d \log^{O(1)} n)$ time. It allows the **ListStragglers** operation to be done in time polynomial in d and $\log n$. This solution does not allow (false) **Delete** x operations that have no matching **Insert** x operations, however. Interestingly, we show that no deterministic algorithm can guarantee correctness in such scenarios, so this drawback should come as no surprise. Nevertheless, we provide a simple randomized solution to the straggler identification problem that uses $O(d \log n \log(1/\epsilon))$ bits and tolerates false deletions, where $\epsilon > 0$ is a user-defined error probability bound. This solution is based on a novel extension to the counting Bloom filter [3, 14], which itself is a dynamic, cardinality-based extension to the well-known Bloom filter data structure [1] (see also [5]). We refer to our extension as the *invertible Bloom filter*, because, unlike the standard Bloom filter and its counting extension—which provide a degree of data privacy protection—the invertible Bloom filter allows for the efficient enumeration of its contents if the number of items it stores is not too large. This might seem like a violation of the spirit of a Bloom filter, which was invented specifically to avoid the space needed for content enumeration. Nevertheless, the invertible Bloom filter is useful for straggler identification, because it can at one time represent, with small space, a multiset that is too large to enumerate, and later, after a series of deletions have been performed, provide for the efficient listing of the remaining elements.

Related Prior Work. Our work is most closely related to the "deterministic k-set structure" of Ganguly and Majumder [16]. This structure solves the straggler detection problem, allowing items to have multiplicity greater than one but disallowing false deletions. This solution, like our deterministic algorithm, is based on finite fields; however the most space-efficient version of their solution uses roughly twice as many bits as ours, and their decoding times are slower: ignoring logarithmic factors, $O(d^3)$ or $O(d^4)$ time, compared to $O(d^2)$ for ours. An additional technical difference is that, for the algorithm of Ganguly and Majumder, the parameter k (analogous to our d) measures the number of distinct stragglers, while for us it measures the total number of stragglers. Independently of our work, Ganguly and Majumder added to the submitted journal version of their paper a lower bound similar to ours proving the impossibility of straggler detection with false deletions (Ganguly, personal communication). Our deterministic solution is also related to work on set reconciliation in communication complexity [23].

Some additional existing work can be adapted to solve the straggler identification problem. For example, Cormode and Muthukrishnan [8] study the problem of identifying the d highest-cardinality members of a dynamic multiset. Their solution can be applied to the straggler identification problem, since whenever there are d or fewer elements in the set, then all elements are of relatively high cardinality. Their result is a randomized data structure that uses $O(d \log^2 n \log(1/\epsilon))$ bits to perform updates in $O(\log^2 n \log(1/\epsilon))$ time and can be adapted to answer **ListStragglers** queries in $O(d \log^2 n \log(1/\epsilon))$ time (in terms of their bit complexities), where $\epsilon > 0$ is a user-defined parameter bounding the probability of a wrong answer.

Also relevant is prior work on combinatorial group testing (CGT), e.g., see [7, 10,11,12,13,15,18,22], and multiple access channels (MAC), e.g., see [6,17,19,20, 21,25,26,28]. In combinatorial group testing, there are d "defective" items in a set U of n objects, for which we are allowed to perform *tests*, which involve forming a subset $T \subseteq U$ and asking if there are any defective items in T. In the standard CGT problem, the outcome is binary—either T contains defective items or it does not. The objective is to identify all d defective items. The CGT algorithms that are most relevant to straggler identification are *nonadaptive*, in that they must ask all of their tests, $T_1, T_2, \ldots, T_m$, in advance. Such an algorithm can be converted to solve the straggler identification problem by creating a counter t_i for each test T_i. On an insertion of x, we would increment each t_i such that $x \in T_i$. Likewise, on a deletion of x, we would decrement each t_i such that $x \in T_i$. The tests with non-zero counters would be exactly those containing our objects of interest, and the nonadaptive CGT algorithm could then be used to identify them. Unfortunately, these algorithms don't translate into efficient straggler-identification methods, as the best known nonadaptive CGT algorithms (e.g., see [11,12]) use $O(d^2 \log n)$ tests, which would translate into a straggler solution needing $O(d^2 \log^2 n)$ bits.

The MAC problem is similar to the CGT problem, except that the items of interest are no longer "defective"—they are d devices, out of a set U, wishing to broadcast a message on a common channel. In this case a "test" is a time slice where members of a subset $T \subseteq U$ can broadcast. Such an event has a three-way outcome, in that there can be 0 devices that use this time slice, 1 device that uses it (in which case it is identified and taken out of the set of potential broadcasters), or there can be 2 or more who attempt to use the channel, in which case none succeed (but all the potential broadcasters learn that T contains at least two broadcasters). Unfortunately, traditional MAC algorithms are adaptive, so do not immediately translate into straggler identification algorithms.

Nevertheless, we can extend the MAC approach further [17,25,26,28], so that each test T returns the actual number of items of interest that are in T. This extension gives rise to a *quantitative* version of CGT (e.g., see [11], Sec. 10.5). Unfortunately, previous approaches to the quantitative CGT problem are either non-constructive [25], adaptive [17,25,26,28], or limited to small values of d. We know of no nonadaptive quantitative CGT algorithms for $d \geq 3$, and the ones for $d = 2$ don't translate into efficient solutions to the straggler identification problem (e.g., see [11], Sec. 11.2).

2 Straggler Detection Via Symmetric Polynomials

We now describe a deterministic algorithm for straggler detection using near-optimal memory. The algorithm is algebraic in nature: it stores as its snapshot of the data stream a collection of *power sums* in a finite field, $GF[p^e]$. The decoding algorithm for this information uses Newton's identities to convert these power sums into the coefficients of a polynomial that has the stragglers as its roots, and finds the roots of this polynomial.

As is standard for this sort of computation, we represent values in $GF[p^e]$ as univariate polynomials of degree at most $e-1$, with coefficients that are integers modulo p; the $GF[p^e]$ arithmetic operations are the standard polynomial arithmetic, modulo a *primitive polynomial* of degree e. Therefore, values in the field $GF[p^e]$ may be represented in space $O(e \log p)$ each. Addition and subtraction of values in $GF[p^e]$ may be performed using modulo-p operations independently over each coefficient, while multiplication of values in $GF[p^e]$ may be performed using a convolution-based polynomial multiplication algorithm, together with reduction modulo the primitive polynomial. Our algorithms also involve division by integers in the range $[2, p-1]$, which may again be done independently on each coefficient. Therefore, each field operation may be performed in bit complexity $\tilde{O}(e \log p)$, where $\tilde{O}(x)$ is a convenient shorthand for $O(x \log^{O(1)} x)$.

Theorem 1. *There is a deterministic streaming straggler detection algorithm using $O(d \log n)$ bits of storage, such that* ***Insert*** *and* ***Delete*** *operations can be performed in bit complexity $\tilde{O}(d \log n)$, and such that* ***ListStragglers*** *operations can be performed in bit complexity $\tilde{O}(d \log^3 n + d^2 \log n + d^{3/2} \log^2 n \min(d, \log n))$.*

Proof. We let p be a prime number, larger than d, and choose e such that $p^e > n$. We perform all operations of the algorithm in the field $GF[p^e]$, and interpret all identifiers in the straggler detection problem as values in this field. The number of bits needed to represent a single value in $GF[p^e]$ is $O(\log n)$, and, with this choice of p and e, each arithmetic operation in the field may be performed in bit complexity $\tilde{O}(\log n)$.

Define the power sums

$$s_k(S) = \sum_{x_i \in S} x_i^k$$

(where x_i and s_k belong to $GF[p^e]$, except for s_0 which we store as a $\log n$ bit integer). Our streaming algorithm stores $s_k(S)$ for $0 \leq k \leq d$. As $s_0(S)$ is the number of stragglers, we can easily compare the number of stragglers to d.

To update the power sums after an insertion of a value x_i, we simply add x_i^k to each power sum s_k; this requires $O(d)$ arithmetic operations in $GF[p^e]$. Similarly, to delete x_i, we subtract x_i^k from each power sum s_k.

At any point in the algorithm, we may define a polynomial in $GF[p^e][x]$,

$$P(x) = \prod_{x_i \in S} (x - x_i) = \sum_{k=0}^{|S|} (-1)^k \sigma_k x^{|S|-k},$$

where σ_k is the kth *elementary symmetric function* of S (the sum of the products of all k-tuples of members of S). These coefficients can be related to the power sums by *Newton's identities* (e.g. see [9]):

$$s_k - k(-1)^k a_k = -\sum_{i=1}^{k-1} (-1)^i \sigma_i s_{k-i}.$$

That is,

$$\begin{aligned}
s_1 - \ \ \sigma_1 &= 0 \\
s_2 + 2\sigma_2 &= \sigma_1 s_1 \\
s_3 - 3\sigma_3 &= \sigma_1 s_2 - \sigma_2 s_1 \\
s_4 + 4\sigma_4 &= \sigma_1 s_3 - \sigma_2 s_2 + \sigma_3 s_1 \\
s_5 - 5\sigma_5 &= \sigma_1 s_4 - \sigma_2 s_3 + \sigma_3 s_2 - \sigma_4 s_1,
\end{aligned}$$

and so on. These equations hold over any field, and in particular over $GF[p^e]$. By using these identities, we may calculate the coefficients of P in sequence from the power sums and the earlier coefficients, using $O(d^2)$ arithmetic operations to compute all coefficients. Note that these calculations involve divisions by the numbers 2, 3, 4, …, d, but all such divisions are possible modulo p. Thus, this stage of the **ListStragglers** operation takes bit complexity $\tilde{O}(d^2 \log n)$.

Finally, to determine the list of stragglers, we find the roots of the polynomial $P(x)$ that has been determined as above. The deterministic root-finding algorithm of Shoup [27] solves this problem in $\tilde{O}(d \log^2 n + d^{3/2} \log n \min(d, \log n))$ field operations; thus, the overall bit complexity bound is as stated. □

We note that a factor of $d^{1/2}$ in Shoup's algorithm [27] occurs only when p has an unexpectedly long repeated subsequence in its sequence of quadratic characters. It seems likely that a more careful choice of p can eliminate this factor, simplifying the time bound for the **ListStragglers** operation to $\tilde{O}(d \log^3 n + d^2 \log n)$. If this is possible, it would be an improvement when d lies in the range of values from $\log^{2/3} n$ to $\log^2 n$.

For $d \leq 4$, the root finding algorithm may be replaced by the usual formulae for solving low degree polynomials in closed form.

3 Impossibility Results for False Deletions

So far, we have assumed that an element deletion can occur only if a corresponding insertion has already occurred. That is, the only anomalous data patterns that might occur are insertions that are not followed by a subsequent deletion. What can we say about more general update sequences in which insertion-deletion pairs may occur out of order, multiple times, or with a deletion that does not match an insertion? We would like to have a streaming data structure that handles these more general event streams and allows us to detect small numbers of anomalies in our insertion-deletion sequences.

Formally, define a *signed multiset* over a set S to be a map f from S to the integers, where $f(x)$ is the number of occurrences of x in the multiset. To insert x into a signed multiset, increase $f(x)$ by one, while to delete x, decrease $f(x)$ by one. Thus, any sequence of insertions and deletions, no matter how ordered, produces a well-defined signed multiset. We wish to find a streaming algorithm that can determine whether all but a small number of elements in the signed multiset have nonzero values of $f(x)$ and identify those elements. But, as we show, for a natural and general class of streaming algorithms, even if restricted to signed multisets in which each x has $f(x) \in \{-1, 0, 1\}$, we cannot distinguish the empty multiset (in which all $f(x)$ are zero) from some nonempty multiset. Therefore, it is impossible for a deterministic streaming algorithm to determine whether a multiset has few nonzeros.

The signed multisets form a commutative group, which we will represent using additive notation: $(f + g)(x) = f(x) + g(x)$. Call this group M. Define a *unit multiset* to be a signed multiset in which all values $f(x)$ are in $\{-1, 0, 1\}$; the unit multisets form a subset of M, but not a subgroup.

Suppose a streaming algorithm maintains information about a signed multiset, subject to insertion and deletion operations. We say that the algorithm is *uniquely represented* if the state of the algorithm at any time depends only on the multiset at that time and not on the ordering of the insertions and deletions by which the multiset was created. That is, there must exist a map u from M to states of the algorithm.

Define a binary operation $+$ on states of a uniquely represented multiset streaming algorithm, as follows. If a and b are states, let A and B be signed multisets such that $u(A) = a$ and $u(B) = b$, and let $a + b = u(A + B)$.

Lemma 1. *If a streaming algorithm is uniquely represented, and $u(P) = u(Q)$, then $u(P + R) = u(Q + R)$.*

Proof. Let s be a sequence of updates that forms R. Then s transforms $u(P)$ to $u(P + R)$ and $u(Q)$ to $U(Q + R)$. Since $u(P) = u(Q)$, $u(P + R)$ must equal $u(Q + R)$. □

Lemma 2. *The operation defined above is well-defined independently of how the representative multisets A and B are chosen, the states of the streaming algorithm form a commutative group under this operation, and u is a group homomorphism.*

Proof. Independence from the choice of representation is Lemma 1. Associativity and commutativity follow from the associativity and commutativity of the corresponding group operation on M. By Lemma 1, $u(A) + u(-A) = u(0)$ and $u(A) + u(0) = u(A)$, so $u(0)$ satisfies the axioms of a group identity; therefore, we have defined a commutative group. That u is a homomorphism follows from the way we have defined our group operations as the images by u of group operations in M. □

Theorem 2. *Any uniquely represented multiset streaming algorithm for a multiset on n items, with fewer than n bits of storage, will be unable to distinguish between the empty set and some nonempty unit multiset.*

Proof. Suppose there are $k < n$ bits of storage, so 2^k possible states. By the pigeonhole principle, two different sets A and B, when interpreted as multisets and mapped to states, map to the same state $u(A) = u(B)$. Then by Lemma 2, $u(A-B) = u(\emptyset)$. $A-B$ is a nonempty unit multiset that cannot be distinguished from the empty set. □

By applying similar ideas, we can prove a similar impossibility result without assumption about the nature of the streaming algorithm.

Theorem 3. *No deterministic streaming algorithm with fewer than n bits of storage can distinguish a stream of matched pairs of insert and delete operations over a set of n items from a stream of insert and delete operations that are not matched in pairs.*

Proof. Suppose that we have a deterministic streaming data structure with $k < n$ bits of storage. For any set A, let $f(A)$ denote the state of the data structure on a stream that starts with an empty set and inserts the items in A in some canonical order. By the pigeonhole principle there exist two sets A and B such that $A \neq B$ but such that $f(A) = f(B)$. Let s_{PQ} $(P, Q \in \{A, B\})$ be the operation stream formed by inserting the items in set P followed by deleting the items in set Q. Then the streaming algorithm must have the same state after stream s_{AA} as it does after stream s_{BA}, but s_{AA} consists of matched insert-delete pairs while s_{BA} does not. □

4 Invertible Bloom Filters

The standard Bloom filter [1] is a randomized data structure for approximately representing a set S subject to insertion operations and membership queries. Given a parameter d on the expected size of S and an error parameter $\epsilon > 0$, it consists of a hash table B containing $m = O(d \log(1/\epsilon))$ single-bit cells (which we denote as a "bit" field), which are initially all 0's, together with $k = \Theta(\log(1/\epsilon))$ random hash functions $\{h_1, \ldots, h_k\}$ that map elements of S to integers in the range $[0, m-1]$. Performing an insert of element x amounts to setting each $B[h_i(x)]$.bit to 1, for $i = 1, \ldots, k$. Likewise, testing for membership of x in S amounts to testing that there is no $i \in \{1, \ldots, k\}$ such that $B[h_i(x)].\texttt{bit} = 0$. Setting the constants appropriately, one can make the probability of returning a false positive to a membership query (that is, an element not in S identified as belonging to S) to be less than ϵ (e.g., see [4]).

The counting Bloom filter [3,14] extends the standard Bloom filter by replacing each "bit" cell of B with a counter cell, "count" (initialized to 0 for each cell). An insertion of item x amounts to incrementing each $B[h_i(x)]$.count by 1, for $i = 1, \ldots, k$. Such a structure also supports the deletion of an item x, by decrementing each cell $B[h_i(x)]$.count by 1, for $i = 1, \ldots, k$. Answering a membership query is similar to that for the standard Bloom filter, amounting to testing that there is no $i \in \{1, \ldots, k\}$ such that $B[h_i(x)].\texttt{count} = 0$.

The *invertible Bloom filter* extends the counting Bloom filter, in several ways, and allows us to solve the straggler identification problem even in the presence of false deletions. It requires that we use three additional random hash functions, f_1, f_2, and g, in addition to the k hash functions, $h_1, \ldots, h_k$, used for B above. The functions, f_1 and f_2 map integers in $[0, n]$ to integers in $[0, m]$. The function g maps integers in $[0, n]$ to integers in $[0, n^2]$. In addition, we add two more fields to each Bloom filter cell, $B[i]$:

- An "idSum" field, which stores the sum of all the elements, x in S, for x's that map to the cell $B[i]$. Note that if $B[i]$ stores m copies of a value x (and no other values), then $B[i].\texttt{idSum} = mx$.
- A "hashSum" field, which stores the sum of all the hash values, $g(x)$, for x's that map to the cell $B[i]$. Note that if $B[i]$ stores m copies of a value x (and no other values), then $B[i].\texttt{hashSum} = mg(x)$.

Moreover, we create a second Bloom filter, C, which has the same number of (count, idSum, and hashSum) fields as B, but uses only the functions f_1 and f_2 to map elements of S to its cells. That is, C is a secondary augmented counting Bloom filter with the same number of cells as B, but with only two random hash functions, f_1 and f_2, to use for mapping purposes. Intuitively, C will serve as a fallback Bloom filter for "catching" elements that are difficult to recover using B alone. Finally, in addition to these fields, we maintain a global count variable, initially 0. Each of our count fields is a signed counter, which (in the case of false deletions) may go negative.

Since all n ID's in U can be represented with $O(\log n)$ bits, their sum can also be represented with $O(\log n)$ bits. Thus, the space needed for B and C is $O(m \log n) = O(d \log n \log(1/\epsilon))$.

We process updates for the invertible Bloom filter as follows.

```
Insert $x$:
    increment count
    for $i = 1, \ldots, k$ do
        increment $B[h_i(x)]$.count
        add $x$ to $B[h_i(x)]$.idSum
        add $g(x)$ to $B[h_i(x)]$.hashSum
    for $i = 1, 2$ do
        increment $C[f_i(x)]$.count
        add $x$ to $C[f_i(x)]$.idSum
        add $g(x)$ to $C[f_i(x)]$.hashSum
Delete $x$:
    decrement count
    for $i = 1, \ldots, k$ do
        decrement $B[h_i(x)]$.count
        subtract $x$ from $B[h_i(x)]$.idSum
        subtract $g(x)$ from $B[h_i(x)]$.hashSum
    for $i = 1, 2$ do
        decrement $C[f_i(x)]$.count
        subtract $x$ from $C[f_i(x)]$.idSum
        subtract $g(x)$ from $C[f_i(x)]$.hashSum
```

That is, to insert x, we go to each cell that x maps to and increment its `count` field, add x to its `idSum` field, and add $g(x)$ to its `hashSum` field. Thus, the methods for element insertion is fairly straightforward. Deletion is similarly easy, in that we simply decrement counts and subtract out the appropriate summands to reverse the insertion operation.

Our method for performing the **ListStragglers** operation is a bit more involved, however. The basic idea is that some cells of B are likely to be *pure*, that is, to have values that have been affected by only a single item. If we can find a pure cell, we can recover the identity of its item by dividing its `idSum` by its `count`. Once a single item and its count are known, we can remove that item from the data structure and continue until all items have been found.

The difficulty with this approach is in finding the pure cells. Because of the possibility of multiple insertions and false deletions, we cannot simply test whether `count` is one: some pure cells may have larger counts (i.e., have multiple copies of the same value), and some impure cells may have a count equal to one (e.g., because of two insertions of a value x followed by a false deletion of a value y that collides with x at this cell). Instead, to test whether a cell is pure, we use its `hashSum`: in a pure cell, the `hashSum` should equal the `count` times the hash of the item's identifier, while in a cell that is not pure it is very unlikely that the `hashSum`, `idSum`, and `count` fields will match up in this way.

The following pseudo-code expresses the decoding algorithm outlined above.

```
**ListStragglers**:
  while $\exists i$, s. t. $g(B[i].\mathtt{idSum}/B[i].\mathtt{count}) = B[i].\mathtt{hashSum}/B[i].\mathtt{count}$ do
    if $B[i].\mathtt{count} > 0$ then {this is a good element}
      Push $x = B[i].\mathtt{idSum}/B[i].\mathtt{count}$ onto an output stack $O$.
      Delete all $B[i].\mathtt{count}$ copies of $x$ from $B$ and $C$ (using a method similar to Delete $x$ above)
    else {this is a false delete}
      Back out all $-B[i].\mathtt{count}$ falsely-removed copies of $x$ from $B$ and $C$ (using a method similar to Insert $x$ above)
  if $\mathtt{count} = 0$ then
    Output the elements in the output stack and insert each element back into $B$ and $C$.
  else {we have mutually-conflicting elements in $B$}
    Repeat the above while loop, but do the tests using $C$ instead of $B$.
    Output the elements in the output stack, $O$, and insert each element back into $B$ and $C$.
```

There is a slight chance that this algorithm fails. For example, we could have two or more items colliding in a cell of B, but we could nevertheless have the condition, $g(B[i].\mathtt{idSum}/B[i].\mathtt{count}) = B[i].\mathtt{hashSum}/B[i].\mathtt{count}$, satisfied (and similarly for C in the second while loop). Fortunately, since g is a random function from $[0, n]$ to $[0, n^2]$, such an event occurs with probability at most $1/n^2$; hence, over the entire algorithm we can assume, with high probability, that it never occurs (since $d << n$). More troubling is the possibility that, even after using the fallback array, C, to find and enumerate elements in the invertible Bloom filter (in the second while loop), we might still have some mutually-conflicting elements in C.

Lemma 3. *If the number of elements in S, which were inserted but not deleted, plus the number of false elements negatively indicated in S, corresponding to items deleted but not inserted, is at most d, then the first while loop will remove all but ϵd such elements from S with probability $1 - \epsilon/2$, for $\epsilon < 1/4$.*

Proof. Omitted due to space limitations. □

Let us assume, therefore, that at most ϵd elements (true and/or false) remain in S after the first while loop. Let us suppose further that each is mapped to two distinct cells in C (the probability there is any such self-collision among the remaining elements in C is at most $\epsilon d/4dk \le \epsilon/4$). We can envision each cell in C as forming a vertex in a graph, and each selected pair of cells as forming an edge in the graph; thus our data can be modeled as a random multigraph with $x \le \epsilon d$ edges and $y = 4dk \ge 8d$ vertices. Thus, it is a very sparse graph. Let $c = y/x \ge 8/\epsilon$.

Two types of bad event could prevent us from decoding the data remaining in C after the first loop. First, two items could map to the same pair of cells, so that our multigraph is not a simple graph. There are $x(x-1)/2$ pairs of items, and each two items collide with probability $2/(y(y-1))$, so the expected number of collisions of this type is $x(x-1)/(y(y-1))$, roughly $1/c^2$. Second, the graph may be simple but may contain a cycle. As shown by Pittel [2, Exercise 8, p. 122], the expected number of vertices in cyclic components of a random graph of this size is bounded by $\sum_{k=3}^{\infty} kc^-k = O(1/c^3)$. Therefore, the expected number of events of either type, and the probability that there exists an event of either type, is $O(1/c^2)$. Choosing $c = O(\sqrt{1/\epsilon})$ is sufficient to show that we will fail in the second while loop with probability at most $\epsilon/4$.

Theorem 4. *If the number of elements in S, which were inserted but not deleted, plus the number of false elements negatively indicated in S, which correspond to items deleted but not inserted, is at most d, then the above algorithm correctly answers a* ***ListStragglers*** *query with probability at least $1-\epsilon$, where $\epsilon < 1/4$.*

Acknowledgments

We would like to thank Dan Hirschberg for several helpful discussions. We are also grateful to an anonymous reviewer for suggesting the multicast application.

References

1. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 422–426 (1970)
2. Bollobás, B.: Random Graphs. Academic Press, New York (1985)
3. Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., Varghese, G.: An improved construction for counting Bloom filters. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 684–695. Springer, Heidelberg (2006)
4. Bose, P., Guo, H., Kranakis, E., Maheshwari, A., Morin, P., Morrison, J., Smid, M., Tang, Y.: On the false-positive rate of Bloom filters. Report, School of Comp. Sci. Carleton Univ. (2007) `http://cg.scs.carleton.ca/~morin/publications/ds/bloom-submitted.pdf`
5. Broder, A., Mitzenmacher, M.: Network applications of Bloom filters: A survey. Internet Mathematics 1(4), 485–509 (2005)
6. Capetanakis, J.I.: Tree algorithms for packet broadcast channels. IEEE Trans. Inf. Theory IT-25(5), 505–515 (1979)
7. Colbourn, Dinitz, Stinson.: Applications of combinatorial designs to communications, cryptography, and networking. In: Walker. (ed.) Surveys in Combinatorics, 1993. London Mathematical Society Lecture Note Series, vol. 187, Cambridge University Press, Cambridge (1999)
8. Cormode, G., Muthukrishnan, S.: What's hot and what's not: tracking most frequent items dynamically. ACM Trans. Database Syst. 30(1), 249–278 (2005)
9. Cox, D., Little, J., O'Shea, D.: Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra. Springer, Heidelberg (1992)
10. DeBonis, A., Gasieniec, L., Vaccaro, U.: Generalized framework for selectors with applications in optimal group testing. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 81–96. Springer, Heidelberg (2003)

11. Du, D.-Z., Hwang, F.K.: Combinatorial Group Testing and Its Applications, 2nd edn. World Scientific, Singapore (2000)
12. Du, D.-Z., Hwang, F.K.: Pooling Designs and Nonadaptive Group Testing. World Scientific, Singapore (2006)
13. Eppstein, D., Goodrich, M.T., Hirschberg, D.S.: Improved combinatorial group testing for real-world problem sizes. In: Dehne, F., López-Ortiz, A., Sack, J.-R. (eds.) WADS 2005. LNCS, vol. 3608, Springer, Heidelberg (2005)
14. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Networking 8(3), 281–293 (2000)
15. Farach, M., Kannan, S., Knill, E., Muthukrishnan, S.: Group testing problems with sequences in experimental molecular biology. In: SEQUENCES, p. 357. IEEE Press, New York (1997)
16. Ganguly, S., Majumder, A.: Deterministic k-set structure. In: Proc. 25th ACM SIGMOD Symp. Principles of Database Systems, pp. 280–289 (2006)
17. Georgiadis, L., Papantoni-Kazakos, P.: A collision resolution protocol for random access channels with energy detectors. IEEE Trans. on Communications COM-30(11), 2413–2420 (1982)
18. Goodrich, M.T., Hirschberg, D.S.: Efficient parallel algorithms for dead sensor diagnosis and multiple access channels. In: 18th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA), pp. 118–127. ACM Press, New York (2006)
19. Greenberg, A.G., Ladner, R.E.: Estimating the multiplicities of conflicts in multiple access channels. In: Proc. 24th Annual Symp. on Foundations of Computer Science (FOCS'83), pp. 383–392. IEEE Computer Society Press, Los Alamitos (1983)
20. Greenberg, A.G., Winograd, S.: A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. J. ACM 32(3), 589–596 (1985)
21. Hofri, M.: Stack algorithms for collision-detecting channels and their analysis: A limited survey. In: Balakrishnan, A.V., Thoma, M. (eds.) Proc. Inf. Sem. Modelling and Performance Evaluation Methodology, Lecture Notes in Control and Info. Sci., vol. 60, pp. 71–85 (1984)
22. Hwang, F.K., Sós, V.T.: Non-adaptive hypergeometric group testing. Studia Scient. Math. Hungarica 22, 257–263 (1987)
23. Minsky, Y., Trachtenberg, A., Zippel, R.: Set reconciliation with nearly optimal communication complexity. IEEE Trans. Information Theory 49(9), 2213–2218 (2003)
24. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
25. Pippenger, N.: Bounds on the performance of protocols for a multiple-access broadcast channel. IEEE Trans. on Information Theory IT-27(2), 145–151 (1981)
26. Ruszinkó, M., Vanroose, P.: A code construction approaching capacity 1 for random access with multiplicity feedback. Report, Fakultät für Mathematik der Universität Bielefeld, Report no. 94-025 (1994) `http://www.math.uni-bielefeld.de/sfb343/preprints/abstracts/apr94025.ps.gz`
27. Shoup, V.: A fast deterministic algorithm for factoring polynomials over finite fields of small characteristic. In: Proc. Int. Symp. Symbolic and Algebraic Computation, pp. 14–21. ACM Press, New York (1991)
28. Tsybakov, B.S.: Resolution of a conflict of known multiplicity. Problems of Information Transmission 16(2), 134–144 (1980)