The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables *

Bernard Chazelle† Joe Kilian‡ Ronitt Rubinfeld‡ Ayellet Tal§

"Oh boy, here is another David Nelson"
Ticket Agent, Los Angeles Airport
(Source: BBC News)

Abstract

We introduce the *Bloomier filter*, a data structure for compactly encoding a function with static support in order to support approximate evaluation queries. Our construction generalizes the classical Bloom filter, an ingenious hashing scheme heavily used in networks and databases, whose main attribute—space efficiency—is achieved at the expense of a tiny false-positive rate. Whereas Bloom filters can handle only set membership queries, our Bloomier filters can deal with arbitrary functions. We give several designs varying in simplicity and optimality, and we provide lower bounds to prove the (near) optimality of our constructions.

1 Introduction

A widely reported news story[1] describes the current predicament facing air passengers with the name of David Nelson, most of whom are being flagged for extra security checks at airports across the United States: "If you think security at airports is tight enough already, imagine your name popping up in airline computers with a red flag as being a possible terrorist. That's what's happening to David Nelsons across the country." The problem is so bad that many David Nelsons have stopped flying altogether. Although the name David Nelson raises a red flag, security officials won't say if there is a terror suspect by that name. "Transportation Security Administration spokesman Nico Melendez said the problem was due to name-matching technology used by airlines."

This story illustrates a common problem that arises when one tries to balance false negatives and false positives: if one is unwilling to accept any false negatives whatsoever, one often pays with a high false positive rate. Ideally, one would like to adjust one's system to fix particularly troublesome false positives while still avoiding the possibility of a false negative (eg, one would like to make life easier for the David Nelsons of the world without making life easier for Osama Bin Laden). We consider these issues for the more prosaic example of Bloom filters, described below.

Historical background *Bloom filters* yield an extremely compact data structure that supports membership queries to a set [1]. Their space requirements fall significantly below the information theoretic lower bounds for error-free data structures. They achieve their efficiency at the cost of a small false positive rate (items not in the set have a small constant probability of being listed as in the set), but have no false negatives (items in the set are always recognized as being in the set). Bloom filters are widely used in practice when storage is at a premium and an occasional false positive is tolerable. They have many uses in networks [2]: for collaborating in overlay and peer-to-peer networks [5, 8, 17], resource routing [15, 26], packet routing [12, 30], and measurement infrastructures [9, 29]. Bloom filters are used in distributed databases to support iceberg queries, differential files access, and to compute joins and semijoins [7, 11, 14, 18, 20, 24]. Bloom filters are also used for approximating membership checking of password data structures [21], web caching [10, 27], and spell checking [22].

Several variants of Bloom filters have been proposed. *Attenuated Bloom filters* [26] use arrays of Bloom filters to store shortest path distance information. *Spectral Bloom filters* [7] extend the data structure to support estimates of frequencies. In *Counting Bloom Filters* [10] each entry in the filter need not be a single bit but rather a small counter. Insertions and deletions to the filter increment or decrement the counters respectively. When the filter is intended to be passed as a message, *compressed Bloom filters* [23] may be used in-

*This work was supported in part by NSF grant CCR-998817, ARO Grant DAAH04-96-1-0181, and NEC Laboratories America.

†Princeton University and NEC Laboratories America, chazelle@cs.princeton.edu

‡NEC Laboratories America, {joe|ronitt}@nec-labs.com

§Technion and Princeton University, ayellet@ee.technion.ac.il

[1] http://news.bbc.co.uk/2/hi/americas/2995288.stm, http://www.kgun9.com/story.asp?TitleID=3201&... ...ProgramOption=News

stead, where parameters can be adjusted to the desired tradeoff between size and false-positive rate.

We note that a standard technique for eliminating a very small number of troublesome false positives is to just keep an exception list. However, this solution does not scale well, both in lookup time and storage, when the list grows large (say comparable to the number of actual positives).

This Work A Bloom filter is a lossy encoding scheme for a set, or equivalently, for the boolean characteristic function of the set. While Bloom filters allow membership queries on a set, we generalize the scheme to a data structure, the *Bloomier filter*, that can encode arbitrary functions. That is, Bloomier filters allow one to associate values with a subset of the domain elements. The method performs well in any situation where the function is defined only over a small portion of the domain, which is a common occurrence. In our (fanciful) terrorist detection example, suspicious names would map to *suspect* and popular, non-suspicious names (eg, David Nelson) would map to *sounds-suspicious-but-really-ok*; meanwhile, all but a tiny fraction of the other names would map to *ok*. This third category is the only source of error. Bloomier filters generalize Bloom filters to functions while maintaining their economical use of storage. In addition, they allow for dynamic updates to the function, provided the support of the function remains unchanged.

Another application of Bloomier filters is to building a *meta-database*, ie, a directory for the union of a small collection of databases. The Bloomier filter keeps track of which database contains information about each entry, thereby allowing the user to jump directly to the relevant databases and bypass those with no relation to the specified entry. Many such meta-databases already exist on the Web: for example, *BibFinder*, a Computer Science Bibliography Mediator which integrates both general and specific search engines; *Debriefing*, a meta search engine that uses results from other search engines, a meta-site for zip codes & postal codes of the world, etc. Bloomier filters can be used to maintain local copies of a directory in any situation in which data or code is maintained in multiple locations.

Our Results Let f be a function from $D = \{0, \ldots, N-1\}$ to $R = \{\perp, 1, \ldots, 2^r - 1\}$, such that $f(x) = \perp$ for all x outside some fixed (arbitrary) subset $S \subseteq D$ of size n. (We use the symbol $\perp$ either to denote 0, in which case the function has support S, or to indicate that f is not defined outside of S.) Bloomier filters allow one to query f at any point of S always correctly and at any point of $D \setminus S$ almost always correctly; specifically, for a random $x \in D \setminus S$, the output returns $f(x) = \perp$ with probability arbitrarily close to 1. Bloomier filters shine especially when the size of D dwarfs that of S, ie, when N/n is very large. The query time is constant and the space requirement is $O(nr)$; this compares favorably with the naive bound of $O(Nr)$, the bound of $O(nr \log N)$ (which is achieved by merely listing the values of all of the elements in the set) and, in the 0/1 case, the $O(n \log \frac{N}{n})$ bound achieved by the perfect hashing method of Brodnik and Munro [3]. (Of course, unlike ours, neither of these methods ever errs.) Bloomier filters are further generalized to handle dynamic updates. One can query and update function values in constant time while keeping the space requirement within $O(nr)$, matching the trivial lower bound to within a constant factor. Specifically, for $x \in S$, we can change the value of $f(x)$, though we cannot change S.

We also prove various lower bounds to show that our results are essentially optimal. First we show that randomization is essential: over large enough domains, linear space is not enough for deterministic Bloomier filters. We also prove that, even in the randomized case, the ability to perform dynamic updates on a changing support (ie, adding/removing x to/from S) requires a data structure with superlinear space.

Our Techniques Our first approach to implementing Bloomier filters is to compose an assortment of Bloom filters into a cascading pipeline. This yields a practical solution, which is also theoretically near-optimal. To optimize the data structure, we change tack and pursue, in the spirit of [4, 6, 19, 28], an algebraic approach based on the expander-like properties of random hash functions.

As with bloom filters, we assume that we can use "ideal" hash functions. We analyze our algorithms in this model; heuristically one can use "practical" hash functions.

2 A Warmup: the Bloom Filter Cascade

We describe a simple, near-optimal design for Bloomier filtering based on a cascading pipeline of Bloom filters. For illustrative purposes, we restrict ourselves to the case $R = \{\perp, 1, 2\}$. Let A (resp. B) be the subset of S mapping to 1 (resp. 2). Note that the "obvious" solution which consists of running the search key x through two Bloom filters, one for A and one for B, does not work: What do we do if both outputs contradict each other? One possible fix is to run the key through a sequence of Bloom filter pairs: $(\mathcal{F}(A_i), \mathcal{F}(B_i))$, for $i = 0, 1, \ldots, \alpha$ and some suitable parameter α. The first pair corresponds to the assignment $A_0 = A$ and $B_0 = B$. Ideally, no key will pass the test for membership in both A and B, as provided by $\mathcal{F}(A_0)$ and $\mathcal{F}(B_0)$, but we cannot count on it. So, we need a second pair of

Bloom filters, and then a third, a fourth, etc. (The idea of multiple Bloom filters appears in a different context in [7].) Generally, we define A_i to be the set of keys in A_{i-1} that pass the test in $\mathcal{F}(B_{i-1})$; by symmetry, B_i is the set of keys in B_{i-1} that pass the test in $\mathcal{F}(A_{i-1})$. In other words, $A_i = A_{i-1} \cap B^*_{i-1}$ and $B_i = B_{i-1} \cap A^*_{i-1}$, where A^*_i and B^*_i are the set of false positives for $\mathcal{F}(A_i)$ and $\mathcal{F}(B_i)$, respectively.

Given an arbitrary key $x \in D$, we run the test with respect to $\mathcal{F}(A_0)$ and then $\mathcal{F}(B_0)$. If one test fails and the other succeeds, we output 1 or 2 accordingly. If both tests fail, we output $\perp$. If both tests succeed, however, we cannot conclude anything. Indeed, we may be faced with two false positives or with a single false positive from either A or B. To resolve these cases, we call the procedure recursively with respect to $\mathcal{F}(A_1)$ and $\mathcal{F}(B_1)$. Note that A_1 (resp. B_1) now plays the role of A (resp. B), while the new universe is $A^* \cap B^*$. Thus, recursively computed outputs of the form '*in* A_1', *in* B_1', '*not in* $A_1 \cup B_1$' are to be translated by simply removing the subscript 1.

For notational convenience, assume that $|A| = |B| = n$. Let n_i be the random variable $\max\{|A_i|, |B_i|\}$. All filters use the same number of hash functions, which is a large enough constant k. The storage allocated for the filters, however, depends on their ranks in the sequence. We provide each of the Bloom filters $\mathcal{F}(A_i)$ and $\mathcal{F}(B_i)$ with an array of size $2^{k^i} k n_i$. The number α of Bloom filter pairs is the smallest i such that $n_i = 0$. A key in A_i ends up in A_{i+1} if it produces a false positive for $\mathcal{F}(B_i)$. This happens with probability at most $(k|B_i|/2^{k^i} k n_i)^k = 2^{-k^{i+1}}$. This implies that a key in A belongs to A_i with probability at most $2^{-(k^{i+1}-k)/(k-1)}$; therefore,

$$\mathbf{E}\, n_i \le n 2^{-(k^{i+1}-k)/(k-1)} \text{ and}$$
$$\mathbf{E}\, \alpha \le 2 \log\log n / \log k.$$

The probability that a search key runs through the i-th filter is less than 2^{-k^i}, so the expected search time is constant. The expected storage used is equal to

$$\mathbf{E} \sum_{i=0}^{\alpha} 2^{k^i} k n_i = kn \sum_{i=0}^{\alpha} 2^{-(k^i-k)/(k-1)} = O(km).$$

Note that, if N is polynomial in n, we can stop the recursion when n_i is about $n/\log n$ and then use perfect hashing [3, 13]. This requires constant time and $O(n)$ bits of extra storage. To summarize, with high probability a random set of hash functions provides a Bloomier filter with the following characteristics: (i) the storage is $O(kn)$ bits; (ii) at most a fraction $O(2^{-k})$ of D produces false positives; and (iii) the search time is $O(\log\log n)$ in the worst case and constant when averaged over all of D.

3 An Optimal Bloomier Filter

Given a domain $D = \{0, \ldots, N-1\}$, a range $R = \{\perp, 1, \ldots, |R|-1\}$, a subset $S = \{t_1, \ldots, t_n\}$ of D, we consider the problem of encoding a function $f : D \mapsto R$, such that $f(t_i) = v_i$ for $1 \le i \le n$ and $f(x) = \perp$ for $x \in D \setminus S$. Note that the function is entirely specified by the *assignment* $A = \{(t_1, v_1), \ldots, (t_n, v_n)\}$. For the purpose of constructing our data structure, we assume that the function values in R are encoded as elements of the additive group $Q = \{0,1\}^q$, with addition defined bitwise mod 2. As we shall see, the false-positive rate is proportional to 2^{-q}, so q must be chosen sufficiently large. Any $x \in R$ is encoded by its q-bit binary expansion ENCODE (x). Conversely, given $y \in Q$, we define DECODE (y) to be the corresponding number if it is less than $|R|$ and $\perp$ otherwise. We use the notation $r = \lceil \log |R| \rceil$.

Given an assignment A, we denote by $A(t)$ the value A assigns to t, ie, $A(t_i) = v_i$. Let Π be a total ordering on S. We write $a >_\Pi b$ to mean that a comes after b in Π. We define $\Pi(i)$ to be the ith element of S in Π; if $i > j$, then obviously $\Pi(i) >_\Pi \Pi(j)$. For any triple (D, m, k), we assume the ability to select a random hash function HASH $: D \to \{1, \ldots, m\}^k$. This allows us to access random locations in a Bloomier filter table of size m.

DEFINITION 3.1. *Given* HASH *as above, let* HASH $(t) = (h_1, \ldots, h_k,)$. *We say that* $\{h_1, \ldots, h_k\}$ *is the* neighborhood *of* t, *denoted* $N(t)$.

Bloomier filter tables store the assignment A, and are created by calling the procedure CREATE $(A)\,[m, k, q]$, where A denotes the assignment and (m, k, q) are the parameters chosen to optimize the implementation. For notational convenience, we will omit mention of these parameters when there is no ambiguity. Our ultimate goal is to create a one-sided error, linear space (measured in bits) data structure supporting constant-bit table lookups. Specifically, we need to implement the following operations:

- CREATE (A): Given an assignment

  $$A = \{(t_1, v_1), \ldots, (t_n, v_n)\},$$

  CREATE (A) sets up a data structure *Table*. The subdomain $\{t_1, \ldots, t_n\}$ specified by A is denoted by S.

- SET_VALUE (t, v, Table): For $t \in D$ and $v \in R$, SET_VALUE (t, v, Table) associates the value v with

domain element t in *Table*. It is *required* that t be in S.

- LOOKUP (t, *Table*): For $t \in S$, LOOKUP (t, *Table*) returns the last value v associated with t. For all but a fraction ε of $D \setminus S$, LOOKUP (t, *Table*) returns $\perp$ (ie, certifies that t is not in S). For the remaining elements of $D \setminus S$, LOOKUP (t, *Table*) returns an arbitrary element of R.

A data structure that supports only CREATE and LOOKUP is referred to as an *immutable* data structure. Note that although re-assignments to elements in S are made by SET_VALUE, no changes to S are allowed. Our lower bounds show that, if we allow S to be modified, then linear size (measured in bits) is impossible to achieve *regardless* of the query time. In other words, Bloomier filters provably rule out fully dynamic operations.

There are three parameters of interest in our constructions: the runtime of each operation, the space requirements, and the false-positive rate ϵ. The CREATE operation runs in expected $O(n \log n)$ time (indeed, $O(n)$ time, depending on the model) and uses $O(n(r + \log 1/\varepsilon))$ space. The SET_VALUE and LOOKUP operations run in $O(1)$ time.

3.1 An Overview We first describe the immutable data structure and later show how to use the same principles to construct a mutable version. The table consists of m q-bit elements, where m and q are implementation parameters. We denote by $Table[i] \in \{0,1\}^q$ the ith q-bit value in *Table*. To look up the value v associated with t, we use a hash function HASH to compute k locations $(h_1, \ldots, h_k)$, where $1 \leq h_i \leq m$, and a q-bit "masking value" M (used for reducing false positives). We then compute $x = M \oplus \bigoplus_{i=1}^{k} Table[h_i]$, where $\oplus$ denotes the bit-wise exclusive-or operation.

There are two main issues to address. First, we must set the values of $Table[i]$, for $i = 1, \ldots, m$, so that the decode operations yield the correct values for all $t \in S$. We need to show that with high probability a "random" solution works (for appropriate parameter settings), and furthermore we wish to compute the assignment efficiently, which we do by a simple greedy algorithm. Second, we must ensure that, for all but an ϵ expected fraction of $t \in D \setminus S$, the computed "image" in Q decodes to $\perp$.

We set the table values using the following key technique. Given a suitable choice of m and k, we show that, with high probability, there is an ordering Π on S and an *order respecting matching*, defined as follows:

DEFINITION 3.2. *Let S be a set with a neighborhood $N(t)$ defined for each $t \in S$. Let Π be a complete ordering on the elements of S. We say that a matching $\mathcal{T}$* respects *(S, Π, N) if (i) for all $t \in S$, $\mathcal{T}(t) \in N(t)$, and (ii) if $t_i >_\Pi t_j$, then $\mathcal{T}(t_i) \notin N(t_j)$. When the function* HASH *(and hence N) is understood from the context, we say that $\mathcal{T}$ respects Π on S.*

Given Π and $\mathcal{T}$, we can, for $t = \Pi(1), \ldots, \Pi(n)$, set the value v associated with t by setting $Table[\mathcal{T}(t)]$. By the order-respecting nature of $\mathcal{T}$, this assignment cannot affect any previously set values. We show the existence of good $(\Pi, \mathcal{T})$ using the notion of *lossless expanders* [6, 28]. Our analysis implies that, with high probability (over HASH), we can find $(\Pi, \mathcal{T})$ in nearly linear time using a greedy algorithm.

To limit the number of false positives, we use the random mask M produced by HASH (t). Because M is distributed uniformly and independently of any of the values stored in *Table*, when we look up $t \notin S$, the resulting value is uniformly and independently distributed over $\{0,1\}^q$. If the size of R is small compared with the size of $\{0,1\}^q$, then with high probability this value will not encode a legal value of R, and we will detect that $t \notin S$.

We make a mutable structure by using a two-table construction. We use the first table, $Table_1$, to encode $\mathcal{T}(t)$ for each $t \in S$. We note that since $\mathrm{N}(t)$ has only k values, which may be computed from HASH (t), $\mathcal{T}(t) \in \mathrm{N}(t)$ can be compactly represented by a number in $\{1, \ldots, k\}$. Now, it follows from the definitions that if $t \neq t'$ for $t, t' \in S$, $\mathcal{T}(t) \neq \mathcal{T}(t')$. Thus, we can simply store the value associated with t in $Table_2[\mathcal{T}(t)]$; the locations will never collide.

3.2 Finding a Good Ordering and Matching We give a greedy algorithm that, given S and HASH, computes a pair $(\Pi, \mathcal{T})$ such that $\mathcal{T}$ respects Π on S. First, we consider how to compactly represent $\mathcal{T}$. Recall that HASH (t) defines the k neighbors, $h_1, \ldots, h_k$ of t. Therefore, given HASH, we can represent $\mathcal{T}(t) \in \mathrm{N}(t)$ by an element of $\{1, \ldots, k\}$. Thus, we define $\iota(t)$ such that $\mathcal{T}(t) = h_{\iota(t)}$. With $S = \{t_1, \ldots, t_n\}$, we also use the shorthand $\iota_i = \iota(t_i)$, from which $\mathcal{T} = \{\iota_1, \ldots, \iota_n\}$. Our algorithm is based on the abundance of "easy matches."

DEFINITION 3.3. *Let $m, k,$* HASH *be fixed, defining $N(t)$ for $t \in D$, and let $S \subseteq D$. We say that a location $h \in \{1, \ldots, m\}$ is a* singleton *for S if $h \in N(t)$ for exactly one $t \in S$. We define* TWEAK $(t, S,$ HASH$)$ *to be the smallest value j such that h_j is a singleton for S, where $N(t) = (h_1, \ldots, h_k)$;* TWEAK $(t, S,$ HASH$) = \perp$ *if no such j exists.*

If TWEAK $(t, S,$ HASH$)$ is defined, then it sets the value of $\iota(t)$ and t is easy to match. Note that this

choice will not interfere with the neighborhood for any different $t' \in S$. Let E denote the subset of S with "easy matches" of that sort, and let $H = S \setminus E$. We recursively find $(\Pi', \mathcal{T}')$ on H and extend $(\Pi', \mathcal{T}')$ to $(\Pi, \mathcal{T})$ as follows. First, we put the elements of E at the end of the ordering for the elements of H, so that if $t \in E$ and $t' \in H$, then $t >_\Pi t'$ (the ordering of the elements within E can be arbitrary). Then we define $\mathcal{T}(t)$ to be the union of the matchings for H and E. It is immediate that $\mathcal{T}$ respects Π on S. We give the algorithm in Figure 1. Note that it is not at all clear that our algorithm for FIND_MATCH will succeed. We show that for m and k suitably large, and HASH chosen at random, FIND_MATCH will succeed with high probability.

3.3 Creating a Mutable Bloomier Filter Given an ordering Π on S, and a matching $\mathcal{T}$ that respects Π on S (given the neighborhoods defined by HASH), we store values associated with any $t \in S$ as follows. Given $t \in S$, $\mathcal{T}$ gives a location $L \in \mathrm{N}(t)$ such that L is not in the neighborhood of any t' that appears before t in Π. Furthermore, given HASH (t), L has a compact description as an element $\ell \in \{1, \ldots, k\}$. Finally, no other $t' \in S$ (before or after t) has the same value of L.

We can construct an immutable table as follows: For $t = \Pi[1], \ldots, \Pi[n]$, we compute the neighborhood $\mathrm{N}(t) = \{h_1, \ldots, h_k\}$ and mask M from HASH (t). From $\mathcal{T}(t)$ we obtain $L \in \mathrm{N}(t)$ with the above properties. Finally, we set $\mathit{Table}[L]$ so that $M \oplus \bigoplus_{i=1}^{k} \mathit{Table}[h_i]$ encodes the value v associated with t. By the properties of L given above, altering $\mathit{Table}[L]$ cannot affect any of the t' whose associated values have already been put into the table. To retrieve the value associated with t, we simply compute

$$x = M \oplus \bigoplus_{i=1}^{k} \mathit{Table}[h_i],$$

and see if x is a correct encoding of some value $v \in R$. If it is not, we declare that $t \notin S$. Because M is random, so is x if $t \notin S$; therefore, it is a valid encoding only with probability $|R|/2^q$.

In order to make a mutable table, we use the fact that each $t \in S$ has a distinct matching value L, with a succinct representation $\ell \in \{1, \ldots, k\}$ (given HASH (t)). We use the above technique to make an immutable table that stores for each t the value ℓ that can be used to recover its distinct matching value L. We then store any value associated with t in the Lth location of a second table.

We give our final algorithms in Figures 2 and 3.

4 Analysis of the Algorithm

The most technically demanding aspect of our analysis is in showing that for a random HASH, and sufficiently large k and m, the FIND_MATCH routine will with high probability find $(\Pi, \mathcal{T})$ such that $\mathcal{T}$ respects Π on S. Once we have such an $(\Pi, \mathcal{T})$, the analysis of our algorithms is straightforward.

LEMMA 4.1. *Assuming that* FIND_MATCH *succeeded in* CREATE, *then for $t \in S$, the value v returned by* LOOKUP (t, Table) *will be the most recent v assigned to t by* CREATE *or* SET_VALUE.

Proof. When the assignment for t is first stored in *Table*, $\mathcal{T}(t)$ generates a location $L \in \mathrm{N}(t)$, with a concise representation $\ell \in \{1, \ldots, k\}$. By the construction, $\mathit{Table}_1[L]$ is set so that

$$M \oplus \bigoplus_{Z \in \mathrm{N}(t)} \mathit{Table}_1[Z]$$

is a valid representation for ℓ. We claim that the same value of ℓ (and hence L) is recovered by the LOOKUP and SET_VALUE commands on input t. These routines recover ℓ by the same formula; it remains to verify that none of the operations causes this value to change. We observe that the LOOKUP and SET_VALUE commands do not alter Table_1. The only indices of Table_1 subsequently altered by CREATE are of the form $\mathcal{T}(t')$, where $t' >_\Pi t$ (since the ts are processed according to Π). However, by the properties of $\mathcal{T}$, it follows that $\mathcal{T}(t') \notin \mathrm{N}(t)$, so these changes to Table_1 cannot affect the recovered value of ℓ, and hence L.

Finally, we observe that all of the L are distinct: Suppose that $t_1, t_2 \in S$ and $t_1 \neq t_2$. Assume without loss of generality that $t_1 >_\Pi t_2$. Then $\mathcal{T}(t_1) \notin \mathrm{N}(t_2)$, but $\mathcal{T}(t_2) \in \mathrm{N}(t_2)$, so $\mathcal{T}(t_1) \neq \mathcal{T}(t_2)$. It follows that $\mathit{Table}_2(L)$ is only altered when CREATE and SET_VALUE associate a value to t, as desired. $\diamondsuit$

LEMMA 4.2. *Suppose that Table is created using an assignment with support S. Then if $t \notin S$,*

$$Pr[\text{LOOKUP}\,(t, \mathit{Table}) = \perp] \geq 1 - \frac{k}{2^q},$$

where the probability is taken over the coins of CREATE, *assuming that* HASH *is a truly random hash function.*

Proof. Since $t \notin S$, the data structures were generated completely independent of the values of

$$(h_1, \ldots, h_k, M) = \text{HASH}\,(t).$$

In particular, M is uniformly distributed over $\{0,1\}^q$, independent of anything else. Hence, the value of

$$M \oplus \bigoplus_{Z \in \mathrm{N}(t)} \mathit{Table}_1[Z]$$

FIND_MATCH (HASH , S)$[m, k]$ — Find $(\Pi, \mathcal{T})$ for S, HASH

1. $E = \emptyset; \Pi = \emptyset$
For $t_i \in S$
  If TWEAK $(t_i, S,$ HASH $)$ is defined
    $\iota_i =$ TWEAK $(t_i, S,$ HASH $)$
    $E = E \cup \{t_i\}$
If $E = \emptyset$ **Return** (**failure**)

2. $H = S \setminus E$
Recursively compute $(\Pi', \mathcal{T}') =$ FIND_MATCH (HASH , H)$[m, k]$.
If FIND_MATCH (HASH ,H)[m,k]=**failure** **Return** (**failure**)

3. $\Pi = \Pi'$
For $t_i \in E$
  Add t_i to the end of Π (ie, make t_i be the largest element in Π thus far)
Return $(\Pi, \mathcal{T} = \{\iota_1, \ldots, \iota_n\})$
(where ι_i is determined for $t_i \in E$, in Step 1, and for $t_i \in H$ (via $\mathcal{T}'$) in Step 2.)

Figure 1: Given HASH and S, FIND_MATCH finds an ordering Π on S and a matching $\mathcal{T}$ on S that respects Π on S.

CREATE $(A = \{(t_1, v_1) \ldots, (t_n, v_n)\})[m, k, q]$ — *(create a mutable table)*

1. Uniformly choose HASH $: D \to \{1, \ldots, m\}^k \times \{0,1\}^q$
$S = \{t_1, \ldots, t_n\}$
Create $Table_1$ to be an array of m elements of $\{0,1\}^q$
Create $Table_2$ to be an array of m elements of R.
(the initial values for both tables are arbitrary)
Put (HASH , m, k, q) into the "header" of $Table_1$
(we assume that these values may be recovered from $Table_1$)

2. $(\Pi, \mathcal{T}) =$ FIND_MATCH (HASH , S)$[m, k]$
If FIND_MATCH (HASH , S)$[m, k] =$ **failure** **Goto** Step 1

3. **For** $t = \Pi[1], \ldots, \Pi[n]$
  $v = A(t)$ (ie, the value assigned by A to t)
  $(h_1, \ldots, h_k, M) =$ HASH (t)
  $L = \mathcal{T}(t); \ell = \iota(t)$ (ie, $L = h_\ell$)

$$Table_1[L] = \text{ENCODE}(\ell) \oplus M \oplus \bigoplus_{\substack{i=1 \\ i \neq \ell}}^{k} Table_1[h_i]$$

  $Table_2[L] = v$

4. **Return** $(Table = (Table_1, Table_2))$

Figure 2: Given an assignment A and parameters m, k, q, CREATE creates a mutable data structure corresponding to A.

LOOKUP $(t, \mathit{Table} = (\mathit{Table}_1, \mathit{Table}_2))$

1. Get (HASH, m, k, q) from Table_1
$(h_1, \ldots, h_k, M) = \text{HASH}(t)$
$\ell = \text{DECODE}\left(M \oplus \bigoplus_{i=1}^{k} \mathit{Table}_1[h_i]\right)$

2. **If** ℓ is defined
$L = h_\ell$
Return $(\mathit{Table}_2[L])$
Else **Return** $(\perp)$

SET_VALUE $(t, v, \mathit{Table} = (\mathit{Table}_1, \mathit{Table}_2))$

1. Get (HASH, m, k, q) from Table_1
$(h_1, \ldots, h_k, M) = \text{HASH}(t)$
$\ell = \text{DECODE}\left(M \oplus \bigoplus_{i=1}^{k} \mathit{Table}_1[h_i]\right)$

2. **If** ℓ is defined
$L = h_\ell$
$\mathit{Table}_2[L] = v$
Return (**success**)
Else **Return** (**failure**)

Figure 3: LOOKUP looks up t in $\mathit{Table} = (\mathit{Table}_1, \mathit{Table}_2)$. SET_VALUE sets the value associated with $t \in S$ to v in $\mathit{Table} = (\mathit{Table}_1, \mathit{Table}_2)$. Note that for SET_VALUE, t must be an element of S, as determined (by A) in the original CREATE invocation.

will be uniformly distributed over $\{0,1\}^q$. However, there are only k out of 2^q values encoding legitimate values of $\ell \in \{1, \ldots, k\}$. $\diamondsuit$

Thus, in the "ideal" setting, where HASH is truly random, it suffices to set

$$q = \left\lceil \lg\left(\frac{k}{\epsilon}\right)\right\rceil.$$

to achieve a false positive rate of ϵ. Of course, HASH is generated heuristically, and is at best pseudorandom, so this analysis must be taken in context. We note that if HASH is a cryptographically strong pseudorandom function, then the behavior of the data structure using HASH will be computationally indistinguishable from that of using a truly random HASH; however, functions with provable pseudorandomness properties are likely to be too complicated to be used in practice.

4.1 Analyzing FIND_MATCH

It remains to show for any S, with reasonable probability (constant probability suffices for our purposes), FIND_MATCH will indeed find a pair $(\Pi, \mathcal{T})$ such that $\mathcal{T}$ respects Π on S. Recall that FIND_MATCH works by finding an "easy" set E, solving the problem recursively for $H = S \setminus E$, and then combining the solutions for E and H to get a solution for S. To show that FIND_MATCH terminates, it suffices to show that for any subset A of S, FIND_MATCH will find a nonempty "easy" set, E_A, implying that FIND_MATCH always makes progress.

Let G be an $n \times m$ bipartite graph defined as follows: On the left side are n vertices, $L_1, \ldots, L_n \in L$, corresponding to elements of S. On the right side are m vertices, $R_1, \ldots, R_m \in R$, corresponding to $\{1, \ldots, m\}$. Recall that HASH defines for each $t_i \in S$ a set of k values $h_1, \ldots, h_k \in \{1, \ldots, m\}$; G contains an edge between L_i and R_{h_j}, for $1 \le j \le k$. The following property is crucial to our analysis:

DEFINITION 4.1. *Let G be as above. We say that G has the* singleton property *if for all nonempty $A \subseteq L$, there exists a vertex $R_i \in R$ such that R_i is adjacent to exactly one vertex in A.*

We claim that if G has the singleton property, then FIND_MATCH will never get stuck. This is because whenever FIND_MATCH is being called on a subset A of S, the resulting easy set will contain R_i, and hence will be nonempty.

We next reduce the singleton property to a well-studied (lossless) expansion property. Let $\mathrm{N}(v)$ be the set of neighbors of a vertex $v \in L$, and for $A \subseteq L$, let $\mathrm{N}(A)$ be the set of neigbors (in R) of elements of A.

DEFINITION 4.2. *Let G be as above. We say that G has the* lossless expansion property *if for all nonempty $A \subseteq L$, $|N(A)| > k|A|/2$.*

LEMMA 4.3. *If G has the lossless expansion property, then it has the singleton property.*

Proof. Assume to the contrary that each node in $\mathrm{N}(A)$ has degree at least 2. Then G graph has at least $2|\mathrm{N}(A)|$ edges. However, by the lossless expansion property, $\mathrm{N}(A) > |A|k/2$, so G has greater than $|A|k$ edges, which is a contradiction.

Now, choosing HASH at random corresponds to choosing G according to the following distribution: Each $v \in L$ selects (with replacement) k random vertices in $r_1, \ldots, r_k \in R$ to be adjacent to. Such random graphs are well studied; Lemma 4.4 follows from a standard counting argument.

LEMMA 4.4. *Let G be chosen as above, with fixed k and $m = ckn$. For any constant $c > 1 + 1/\sqrt{n}$ and n sufficiently large, G has the lossless expansion probability with probability at least $2/3$.*

Proof. (Sketch) The probability of a counterexample is at most

$$\begin{aligned}
&\sum_{s=1}^{n} \binom{n}{s} \binom{m}{\lfloor ks/2 \rfloor} \left(\frac{\lfloor ks/2 \rfloor}{m}\right)^{ks} \\
&\approx \sum_{s=1}^{n} \left(\frac{en}{s}\right)^{s} \left(\frac{2ecn}{s}\right)^{ks/2} \left(\frac{s}{2cn}\right)^{ks} \\
&\approx \sum_{s=1}^{n} \left(\frac{e^{k/2+1}}{2^{k/2}s}\right)^{s} \left(\frac{s}{cn}\right)^{ks/2} \\
&\leq o(1) + \sum_{s \geq \sqrt{n}} c^{-s} \\
&= o(1).
\end{aligned}$$

$\diamondsuit$

5 Lower Bounds

We consider the case $R = \{\perp, 1, 2\}$. The set S splits into the subsets A and B that map to 1 and 2, respectively. It is natural to wonder whether a single set of hash functions might be sufficient for all pairs A, B. In other words, is deterministic Bloomier filtering possible with only $O(n)$ bits of storage. We provide a negative answer to this question. Our lower bound also holds for nonuniform algorithms; in other words, we may use an arbitrary large amount of storage besides the Bloomier filter tables as long as the encoding depends only on n and N.

THEOREM 5.1. *Deterministic Bloomier filtering requires $\Omega(n + \log\log N)$ bits of storage.*

Proof. Let G be a graph with $\binom{N}{2n}\binom{2n}{n}$ nodes, each one corresponding to a distinct vector $\{-1, 0, 1\}^N$ with exactly n coordinates equal to 1 and n others equal to -1. Two nodes of G are adjacent if there exists at least one coordinate position $1 \leq i \leq N$ such that their corresponding vectors $(x_1, \ldots, x_N)$ and $(y_1, \ldots, y_N)$ satisfy $x_i y_i = -1$. Intuitively, each node corresponds to a choice of A (the 1 coordinates) and B (the -1 coordinates). Two nodes are joined by an edge if the set A of one node intersects the set B of the other one. Since the table T is the only source of information about A, B, no two adjacent nodes should correspond to the same table assignment; therefore, the size m of the array is at least $\log \chi(G)$, where $\chi(G)$ is the chromatic number of G.

Theorem 5.1 follows directly from the lemma below.

$\diamondsuit$

LEMMA 5.1. *The chromatic number of G is between $\Omega(2^n \log N)$ and $O(4^n \ln N)$.*

Proof. Consider a minimum coloring of G. For any vector $w \in \{-1, 1\}^n$ and any sequence of n indices $1 \leq i_1 < \cdots < i_n \leq N$, there exists a color c such that $w = (z^c_{i_1}, \ldots, z^c_{i_n})$. To see why, consider a choice of A, B that matches the coordinates of w at the positions $i_1, \ldots, i_n$. If we turn all the minus ones to zeroes, the resulting set of vectors z^c is (N, n)-universal (meaning that the restrictions of the vectors z^c to any given choice of n coordinate positions produce all possible 2^n patterns). By Kleitman and Spencer [16], the number of such vectors is known to be $\Omega(2^n \log N)$. For the upper bound, we use the existence of a $(N, 2n)$-universal set of vectors of size $O(n2^n \log N)$—also established in [16]—and turn all zeroes into minus ones. (Alternatively, we can use Razborov's bound on the size of separating sets [25].) Each node is colored by picking a vector from the universal set that matches the the ones and minus ones of the vector associated with that node. $\diamondsuit$

Going back to the randomized model of Bloomier filtering, we consider what happens if we attempt to modify the set S itself. Again we give a negative answer, but this time the universe size need be only polynomially larger than n for the scheme to break down. Intuitively, this shows that too much information is lost in a linear bit size encoding of the function f to allow for changes in the set S.

THEOREM 5.2. *If $N = 2^{n^{O(1)}}$ and the number m of storage bits satisfies $n \leq m \leq \frac{n}{c} \log\log(N/cn^3)$ for some large enough constant c, then Bloomier filtering cannot support dynamic updates on the set S.*

Proof. Again, we consider S to be the disjoint union of of an n-set A (resp. B) mapping to 1 (resp. 2). Fix the original B, and consider the assignments of the table T corresponding to the various choices of A. With each assignment of T is associated a certain family of n-element sets $A \subseteq D$. Let $\mathcal{F}$ be the largest such family: obviously, $|\mathcal{F}| \geq \binom{N}{n} 2^{-m}$. Given an integer $k > 0$, let L_k be the set of elements $x \in U$ that belong to at least k sets of $|\mathcal{F}|$. It is easy to see that L_k cannot be too small. Indeed, let $\mathcal{F}_k$ denote the subfamily consisting of the sets of $\mathcal{F}$ that are subsets of L_k. Obviously, $|\mathcal{F} \setminus \mathcal{F}_k| \leq (k-1)N$. The assumptions of the theorem imply that $N > cn^3$; thus, the choice of $k = \lfloor |\mathcal{F}|/2N \rfloor$ ensures that $k > c$ and

$$|\mathcal{F}_k| \geq |\mathcal{F}| - (k-1)N > \frac{1}{2}|\mathcal{F}| \geq \binom{N}{n} 2^{-m-1}.$$

Because $\binom{L_k}{n} \geq |\mathcal{F}_k|$, it follows that

$$|L_k| \geq 2^{-\frac{m+1}{n}} N. \tag{5.1}$$

The expected number of sets in $\mathcal{F}$ that a random element from D intersects is $\frac{n}{N}|\mathcal{F}|$. Given an n-element $B \subseteq D$, let $\mathcal{F}^B$ denote the subfamily of $\mathcal{F}$ whose sets intersect B. For a random B,

$$\mathbf{E} \sum \Big\{ |S| \,:\, S \in \mathcal{F}^B \Big\} \leq \frac{n^3}{N} |\mathcal{F}|.$$

Let $\mathcal{F}_c^B = \mathcal{F} \setminus \mathcal{F}^B$ and $L_k^B = \bigcup \{ S \cap L_k \,|\, S \in \mathcal{F}_c^B \}$. Since each $x \in L_k$ intersects at least k sets of $\mathcal{F}$,

$$\mathbf{E}\, |L_k^B| \geq |L_k| - \frac{n^3}{kN} |\mathcal{F}| \geq |L_k| - 3n^3.$$

Once the new choice of B is revealed, the table gets updated. The only information about A is encoded in the previous table assignment. Thus the algorithm cannot distinguish between any two sets A in $\mathcal{F}_c^B$. To summarize, given a random B, the algorithm must answer '*in* A' for any search key in L_k^B; furthermore,

$$\text{Prob}\Big[\, |L_k^B| \geq |L_k| - 6n^3 \Big] \geq \frac{1}{2}. \tag{5.2}$$

Next, we partition the family of n-element sets B according to the assignment of T each one corresponds to. This gives us at most 2^m subfamilies $\{\mathcal{G}_i\}$, with $\sum_i |\{\mathcal{G}_i\}| = \binom{N}{n}$. If M_i denotes the union $\bigcup \{ S \cap L_k \,|\, S \in \mathcal{G}_i \}$, then given any new choice of B in $\mathcal{G}_i$ the algorithm must answer '*in* B' for any search key in M_i. By our previous remark, therefore, it is imperative that M_i should be disjoint from L_k^B. We show that if m is too small, then for a random choice of B both sets intersect with high probability.

Fix a parameter $\lambda = \lfloor |L_k| 2^{-c^{m/n}} \rfloor$. Let $i(B)$ denote the index j such that $B \in \mathcal{G}_j$ and let $\tau = \lceil |L_k| n/2N \rceil$. Given a random B, by Chernoff's bound, the probability that $|B \cap L_k| < \tau$ is $o(1)$. On the other hand, the conditional probability that $|M_{i(B)}| \leq \lambda$, given that $|B \cap L_k| \geq \tau$, is at most

$$\begin{aligned} \max_{s \geq \tau} 2^m \binom{\lambda}{s} \Big/ \binom{|L_k|}{s} &= 2^m \binom{\lambda}{\tau} \Big/ \binom{|L_k|}{\tau} \\ &\leq 2^m \frac{\lambda^\tau}{|L_k|^\tau} \\ &= o(1). \end{aligned}$$

Therefore, the probability that $|M_{i(B)}| > \lambda$ is $1 - o(1)$. Since $\lambda > 6n^3$, it follows that, with probability at least $1/2 - o(1)$, $|M_{i(B)}|$ intersects L_k^B and the algorithm fails. $\diamondsuit$

References

[1] Bloom, B. *Space/time tradeoffs in in hash coding with allowable errors*, CACM 13 (1970), 422–426.

[2] Broder, A., Mitzenmacher, M. *Network applications of Bloom filters: a survey*, Allerton 2002.

[3] Brodnik, A., Munro, J.I., *Membership in constant time and almost minimum space*, SIAM J. Comput. 28 (1999), 1628–1640.

[4] Buhrman, H., Miltersen, P.B., Radhakrishnan, J., Venkatesh, S. *Are bitvectors optimal?* Proc. 32th STOC (2000), 449–458.

[5] Byers, J., Considine, J., Mitzenmacher, M. *Informed content delivery over adaptive overlay networks*, Proc. ACM SIGCOMM 2002, Vol. 32:4, Computer Communication Review (2002), 47–60.

[6] Capalbo, M., Reingold, O., Vadhan, S., Wigderson, A. *Randomness conductors and constant-degree expansion beyond the degree /2 barrier*, Proc. 34th STOC (2002), 659–668.

[7] Cohen, S., Matias, Y. *Spectral Bloom filters*, SIGMOD 2003.

[8] Cuena-Acuna, F.M., Peery, C., Martin, R.P., Nguyen, T.D. *PlanetP: Using gossiping to build content addressable peer-to-peer information sharing communities*, Rutgers Technical Report DCS-TR-487, 2002.

[9] Estan, C., Varghese, G. *New directions in traffic measurement and accounting*, Proc. ACM SIGCOMM 2002, Vol 32:4, Computer Communication Review (2002), 323–336.

[10] Fan, L., Cao. P., Almeida, J., Broder, A. *Summary cache: a scalable wide-area web cache sharing protocol*, IEEE / ACM Transactions on Networking, 8 (2000), 281-293.

[11] Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., Ullman. J. *Computing iceberg queries efficiently*, Proc. 24th Int. Conf. on VLDB (1998), 299–310.

[12] Feng, W.-C., Shin, K.G., Kandlur, D., Saha, D. *Stochastic fair blue: A queue management algorithm for enforcing fairness*, INFOCOM '01 (2001), 1520–1529.

[13] Fredman, M.L., Komlos, J., Szemeredi, E. *Storing a sparse table with O(1) worst case access time*, J. ACM 31 (1984), 538–544.

[14] Gremillion, L.L. *Designing a Bloom Filter for differential file access*, Comm. ACM 25 (1982), 600–604.

[15] Hsiao, P. *Geographical region summary service for geographical routing*, Mobile Computing and Communications Review 5 (2001), 25–39.

[16] Kleitman, D.J., Spencer, J. *Families of k-independent sets*, Discrete Math 6 (1973), 255–262.

[17] Ledlie, J., Taylor, J., Serban, L., Seltzer, M. *Self-organization in peer-to-peer systems*, Proc. 10th European SIGOPS Workshop, September 2002.

[18] Li, Z., Ross, K.A. *PERF join: an alternative to two-way semijoin and bloomjoin* CIKM '95, Proc. 1995 In-

ternational Conference on Information and Knowledge Management, 137–144, November 1995.

[19] Luby, M. *LT codes*, Proc. 43rd Annu. IEEE Symp. Foundat. Comput. Sci., 2002.

[20] Mackert, L., Lohman, G. *R* optimizer validation and performance for distributed queries*, Proc. Int'l. Conf. on VLDB (1986), 149–159.

[21] Manber, U., Wu, S. *An algorithm for approximate membership checking with application to password security*, Information Processing Letters 50 (1994), 191–197.

[22] McIlroy, M.D. *Development of a spelling list*, IEEE Trans. on Communications, COM-30 (1982), 91–99.

[23] Mitzenmacher, M. *Compressed Bloom filters*, IEEE Transactions on Networking 10 (2002).

[24] Mullin, J.K. *Optimal semijoins for distributed database systems*, IEEE Transactions on Software Engineering 16 (1990).

[25] Razborov, A.A. *Applications of matrix methods to the theory of lower bounds in computational complexity*, Combinatorica 10 (1990), 81–93.

[26] Rhea, S.C., Kubiatowicz, J. *Probabilistic location and routing*, Proceedings of INFOCOM 2002.

[27] Rousskov, A., Wessels, D. *Cache digests*, Computer Networks and ISDN Systems, 30(22-23), 2155-2168, 1998.

[28] Sipser, M., Spielman, D.A. *Expander codes*, IEEE Trans. Inform. Theory 42 (1996), 1710–1722.

[29] Snoeren, A.C., Partridge, C., Sanchez, L.A., Jones, C.E., Tchakountio, F., Kent, S.T., Strayer W.T. *Hash-based IP traceback*, Proc. ACM SIGCOMM 2001, Vol. 31:4, Computer Communication Review, 3–14, August 2001.

[30] Whitaker, A., Wetherall, D. *Forwarding without loops in Icarus*, Proc. 5th OPENARCH (2002), 63–75.