Simple Summaries for Hashing With Choices

Adam Kirsch, *Student Member, IEEE*, and Michael Mitzenmacher, *Member, IEEE*

***Abstract*—In a multiple-choice hashing scheme, each item is stored in one of $d \geq 2$ possible hash table buckets. The availability of these multiple choices allows for a substantial reduction in the maximum load of the buckets. However, a lookup may now require examining each of the d locations. For applications where this cost is undesirable, Song *et al.* propose keeping a summary that allows one to determine which of the d locations is appropriate for each item, where the summary may allow false positives for items not in hash table. We propose alternative, simple constructions of such summaries that use less space for both the summary and the underlying hash table. Moreover, our constructions are easily analyzable and tunable.**

***Index Terms*—Hash tables, router architecture, table lookup.**

I. Introduction

In a multiple-choice hashing scheme, a hash table is built using the following approach: each item x is associated with hash values $h_1(x), h_2(x), \ldots, h_d(x)$, each corresponding to a bucket in the hash table, and the item is placed in one (or possibly more) of the d locations. Such schemes are often used to lessen the maximum load (that is, the number of items in a bucket), as giving each item the choice between more than one bucket in the hash table often leads to a significant improvement in the balance of the items [1], [4], [14], [17]. These schemes can also be used to ensure that each bucket contains at most one item with high probability [3]. For these reasons, multiple-choice hashing schemes have been proposed for many applications, including network routers [4], peer-to-peer applications [6], and standard load balancing of jobs across machines [15].

Recently, in the context of routers, Song *et al.* [20] suggested that a drawback of multiple-choice schemes is that at the time of a lookup, one cannot know which of the d possible locations to check for the item. The natural solution to this problem is to use d lookups in parallel [4]. But while this approach might keep the lookup time the same as in the standard single-choice hashing scheme, it generally costs in other resources, such as pin count in the router setting. Song *et al.* [20] provide a framework for avoiding these lookup-related costs while still allowing insertions and deletions of items in the hash table. They suggest keeping a small *summary* in very fast memory (that is, significantly faster memory than is practical to store the much larger hash table) that can efficiently answer queries of the form: "Is x in the hash table, and if so, which of $h_1(x), \ldots, h_d(x)$ was actually used to store x?" A simple illustration is given in Fig. 1. Of course, items that are not actually in the hash table may yield false positives; otherwise, the summary could be no more efficient than a hash table. The small summary used by Song *et al.* [20] consists of a counting Bloom filter [10], [16]. We review this construction in detail in Section IV.

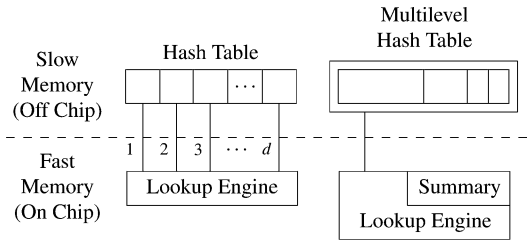

Fig. 1. A comparative illustration of the hardware and connections in an implementation of a standard multiple choice hash table (left) and a *multilevel hash table* (described in Section VI) equipped with a summary data structure (right).

The specific applications that motivate this work are a wide variety of packet processing techniques employed by high-speed routers that rely on the use of a hash table. In these applications, the worst-case performance of the standard lookup, insertion, and even deletion operations can be a bottleneck. In particular, hash tables that merely guarantee good amortized performance may not work well in these settings because it may not be practical to allow any exceptionally time-consuming operations, even if they are rare, since that might require buffering the operations that arrive in the interim. Furthermore, the high cost of computational, hardware, and even energy resources in this setting forces us to seek hash table optimizations that are not only effective, but also extremely amenable to simple, efficient, hardware-based implementations. These observations explain why both we and Song *et al.* focus on the use of multiple-choice hash tables, prized for their worst-case performance guarantees, along with a summary data structure that allows for a further reduction in the cost of lookups into the hash table.

In this paper, we suggest three alternative approaches for maintaining summaries for multiple-choice hashing schemes. The first is based on interpolation search, and the other two are based on standard Bloom filters or simple variants thereof and a clever choice of the underlying hash table. Our approaches have numerous advantages, including less space for the hash table, similar or smaller space for the summary, and better performance for insertions and deletions. Another advantage of our approaches is that they are very simple to analyze; while [20] provides an analysis for a basic variation of the scheme proposed therein, more advanced versions (which seem to be required for adequate performance) have not yet been analyzed. We believe that the ability to analyze performance is important,

Manuscript received December 20, 2005; revised August 18, 2006; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Buddhikot. This work was supported in part by NSF Grant CCR-0121154 and a grant from Cisco Systems. The work of A. Kirsch was also supported in part by an NSF Graduate Research Fellowship. A preliminary version of this work appeared in the Proceedings of the 43rd Annual Allerton Conference on Communication, Control, and Computing, 2005.

A. Kirsch and M. Mitzenmacher are with the School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138 USA (e-mail: kirsch@eecs.harvard.edu; michaelm@eecs.harvard.edu)

Digital Object Identifier 10.1109/TNET.2007.899058

as it allows for more informed engineering decisions based on formal guarantees.

An interesting feature of our work is that we use multiple-choice hashing schemes that are sharply *skewed*, in the sense that most items are placed according the first hash function, fewer are placed according to the second, and so on. We show how to take advantage of this skew, providing an interesting principle: dividing the hash table space unequally among the d sub-tables allows for skew tradeoffs that can allow significant performance improvements for the corresponding summary.

II. Related Work

There is a lot of work on multiple-choice hashing [17] and on Bloom filters [2], [5]; for readers unfamiliar with Bloom filters, we give a brief review in Section III. Our primary starting point is the work of Song *et al.* [20], which introduces an approach for summarizing the locations of items in a hash table that uses multiple hash functions. We review this work in detail in Section IV.

We are also influenced by a significant early paper of Broder and Karlin [3]. For n items, they give a construction of a *multilevel hash table* (MHT) that consists of $d = O(\log\log n)$ sub-tables, each with its own hash function, that are geometrically decreasing in size. An item is always placed in the first sub-table where its hash location is empty, and therefore this hashing scheme is skewed in the sense described above. Our main result can be seen as adding a summary to an MHT, and so we give a more thorough description of MHTs in Section VI.

Finally, the problem of constructing summaries for multiple-choice hash tables seems closely connected with the work on a generalization of Bloom filters called *Bloomier filters* [7], which are designed to represent functions on a set. In the Bloomier filter problem setting, each item in a set has an associated value; items not in the set have a null value. The goal is then to design a data structure that gives the correct value for each item in the set, while allowing *false positives*, so that a query for an item not in the set may rarely return a non-null value. In our setting, values correspond to the hash function used to place the item in the table. For static hash tables (that is, with no insertions or deletions), current results for Bloomier filters could be directly applied to give a (perhaps inefficient) solution. Limited results exist for Bloomier filters that have to cope with changing function values. However, lower bounds for such filters [7], [19] suggest that we must take advantage of the characteristics of our specific problem setting (e.g., the skew of the distribution of the values) in order to guarantee good performance.

III. Bloom Filter Review

In this section, we briefly review the fundamentals of Bloom filters, based on the presentation of the survey [5]. A Bloom filter for representing a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements from a large universe U consists of an array of m bits, initially all set to 0. The filter uses k (independent, random) hash functions $g_1, \ldots, g_k$ with range $\{1, \ldots, m\}$. For each element $x \in S$, the bits $g_i(x)$ are set to 1 for $1 \leq i \leq k$. To check if an item y is in S, we check whether all $g_i(y)$ are set to 1. If not, then clearly y is not a member of S. If all $g_i(y)$ are set to 1, we assume that y is in S, and hence a Bloom filter may yield a *false positive*.

The probability that some $y \notin S$ yields a false positive, or the *false positive probability*, is easy to analyze. After all the elements of S are hashed into the Bloom filter, the probability that a specific bit is still 0 is

$$p' = (1 - 1/m)^{kn} \approx e^{-kn/m}.$$

In this section, we generally use the approximation $p = e^{-kn/m}$ in place of p' for convenience.

If ρ is the proportion of 0 bits after all the n elements are inserted in the table, then conditioned on ρ the probability of a false positive is

$$(1 - \rho)^k \approx (1 - p')^k \approx (1 - p)^k = (1 - e^{-kn/m})^k.$$

These approximations follow since $\mathbf{E}[\rho] = p'$, and ρ can be shown to be highly concentrated around p' using standard techniques. It is easy to show that the expression $(1 - e^{-kn/m})^k$ is minimized when $k = (m/n)\ln 2$ (neglecting the integrality of k), giving a false positive probability f of

$$f = (1 - e^{-kn/m})^k = (1/2)^k \approx (0.6185)^{m/n}.$$

Finally, we note that sometimes Bloom filters are described slightly differently, with each hash function having a disjoint range of m/k consecutive bit locations instead of having one shared array of m bits. We refer to this variant as a *partitioned* Bloom filter. Repeating the analysis above, we find that in this case the probability that a specific bit is 0 is

$$(1 - k/m)^n \approx e^{-kn/m},$$

and so, asymptotically, the performance is the same as the original, unpartitioned scheme.

IV. The Scheme of Song *et al.* [20]

For comparison, we review the summary of [20] before introducing our approaches. The basic scheme works as follows. The hash table consists of m *buckets*, and each item is hashed via d (independent, random) hash functions to d buckets (with multiplicity in the case of collisions). The summary consists of one b-bit *counter* for each bucket. Each counter tracks the number of item hashes to its corresponding bucket.

In the static setting (where the hash table is built once and never subsequently modified), all n items are initially hashed into a preliminary hash table and each is stored in all of its hash buckets. After all items have been hashed, the table is *pruned*; for each item, the copy in the hash bucket with the smallest counter (breaking ties according to bucket ordering) is kept, and all other copies are deleted. Determining the location of an item in the table is now quite easy: one need only compute the d buckets corresponding to the item and choose the one whose corresponding counter is minimal (breaking ties according to bucket ordering). Of course, when looking up an item not in the hash table, there is some chance that all of the examined counters are nonzero, in which case the summary yields a *false positive*. That is, the item appears to be in the hash table until the table is actually checked at the end of the lookup procedure.

While asymptotics are not given in [20], Song *et al.* strive for parameters that guarantee that there is at most 1 item per bucket with high probability, although their approach could be

used more generally. Under this constraint, as we show below, one can achieve $O(n(\log n)\log\log n)$ bits in the summary with $m = \Theta(n\log n), d = \Theta(\log n)$, and $b = \Theta(\log\log n)$; we suspect this bound is tight.

Since the fraction F of nonempty buckets in the hash table before the pruning step satisfies $\mathbf{E}[F] = 1 - (1 - 1/m)^{nd} = 1 - \Omega(1)$, we have $F > (1+\epsilon)\mathbf{E}[F]$ with probability at most $n^{-\Omega(n)}$, for any constant $\epsilon > 0$ (by a standard martingale argument). This observation tells us that if an $(n+1)$-st item is inserted into the hash table just before the pruning step, the probability that all d of its hash buckets are already occupied is at most $n^{-\Omega(n)} + [(1+\epsilon)\mathbf{E}[F]]^d \leq n^{-c}$ for sufficiently small $\epsilon > 0$, any constant c, sufficiently large n, and some $d = \Theta(\log n)$. This probability is clearly an upper bound on the probability that any particular item hashes to an already occupied bucket. Taking a union bound over all n items tells us that even before pruning, the maximum load is 1 with high probability. Finally, a standard balls-and-bins result (e.g., [18, Lemma 5.1]) tells us that the maximum value of the counters is $O(\log n)$ with high probability, so we may choose $b = \Theta(\log\log n)$.

This basic scheme is not particularly effective. To improve it, Song *et al.* give heuristics to remove collisions in the hash table. The heuristics appear effective, but they are not analyzed. Insertions can be handled readily, but can require relocating previously placed items. Song *et al.* show that the *expected* number of relocations per insertion is constant, but they do not give any high probability bounds on the number of relocations required. Deletions are significantly more challenging under this framework, necessitating additional data structures beyond the summary that require significant memory (for example, a copy of the unpruned hash table, where each item is stored in all of corresponding hash buckets, or a smaller variation called a Shared-node Fast Hash Table) and possibly time; see [20] for details.

V. Separating Hash Tables and Their Summaries

Following Song *et al.* [20], we give the following list of goals for our hash table and summary constructions:

- Achieving a maximum load of 1 item per hash table bucket with high probability. (All of our work can be generalized to handle any fixed constant maximum load.)
- Minimizing space for the hash table.
- Minimizing space for the summary.
- Minimizing false positives (generated by the summary) for items not in the hash table.
- Allowing insertions and deletions to the hash table, with corresponding updates for the summary.

As a first step, we suggest the following key idea: the summary data structure *need not* correspond to a counter for each bucket in the hash table. That is, we wish to separate the format of the summary structure from the format of the hash table. This change allows us to optimize the hash table and the summary individually. We exploit this additional flexibility in the specific constructions that we present.

The cost of separating the hash table and the summary is additional hashing. With the approach of [20], the hashes of an item are used to access both the summary and the hash table. By separating the formats of the summary and the hash table, we must also separate the roles of the hash functions, and so we must introduce extra hash functions and computation. However, hashing computation is unlikely to be a bottleneck resource in the applications we have in mind, where all of the computations are done in parallel in hardware. Thus, the costs seem reasonable compared to what we will gain, particularly in storage requirements.

A further small disadvantage of separating the summary structure from the hash table is that the summaries we suggest do not immediately tell us if a hash table bucket is empty or not, unlike the summary of [20]. To handle insertions and deletions easily, we therefore require a bit table with one bit per bucket to denote whether each bucket contains an item. Strictly speaking this table is not necessary – we could simply check buckets in the hash table when needed – but in practice this would be inefficient, and hence we add this cost in our analysis of our summary structures.

Having decided to separate the summary structure from the underlying hash table, the next design issue is what underlying hash table to use. We argue that the multilevel hash table of Broder and Karlin [3] offers excellent performance with very small space overhead.

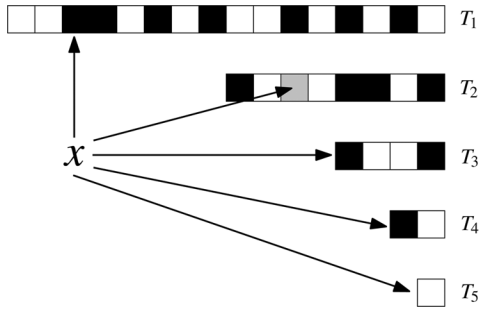

Fig. 2. Inserting an item x into an MHT. The lines represent the hash locations of x, the black cells represent occupied buckets, the white cells represent unoccupied buckets, and the grey cell represents the bucket in which x is placed. In this example, we place x in $T_2[h_2(x)]$ since that bucket is empty and $T_1[h_1(x)]$ is occupied.

VI. The Multilevel Hash Table (MHT) [3]

The multilevel hash table (MHT) for representing a set of n elements consists of $d = \log\log n + O(1)$ sub-tables, $T_1, \ldots, T_d$, where T_i has $c_2^{i-1}c_1 n$ buckets that can each hold a single item, for some constant parameters $c_1 > 1$ and $c_2 < 1$. Since the T_i's are geometrically decreasing in size, the total size of the table is linear (it is bounded by $c_1 n/(1 - c_2)$). For simplicity, we assume that $T_1, \ldots, T_d$ place items using independent fully random hash functions $h_1, \ldots, h_d$.

To place an item x in the MHT, we simply find the smallest i for which $T_i[h_i(x)]$ is empty, and place x at $T_i[h_i(x)]$. We illustrate this procedure in Fig. 2. Of course, this scheme relies on the assumption that at least one of the $T_i[h_i(x)]$'s will always be empty. Following [3], we say that a *crisis* occurs if this assumption fails. By slightly modifying the original analysis, we can show that when $c_1 c_2 > 1$, the probability that there is any

crisis is polynomially small.[1] The formal statement and proof of our result can be found in Appendix I.

Finally, we note that deletions can be handled by simply removing items from the hash table. While theoretical bounds are harder to come by when deletions occur, related work shows that multiple-choice hashing still does very well [17].

A. Approximate and Exact Calculations for MHTs

Given a specific configuration of the MHT and the number of items n, one can easily approximate and exactly calculate the probability that a crisis occurs during the sequential insertion of n items into an initially empty MHT. The ability to calculate such numbers is important for making appropriate engineering decisions and is a useful feature of this approach.

A simple approximate calculation can be made by using expectations. In general, if we attempt to hash αn items into a sub-table of size βn, the expected number of nonempty buckets (which is the same as the expected number of items placed in the sub-table) is

$$\beta n[1-(1-1/\beta n)^{\alpha n}] \geq \beta n(1-\exp(-\alpha/\beta)). \quad (1)$$

Thus the expected number of items left to be placed in subsequent sub-tables is

$$\alpha n - \beta n[1-(1-1/\beta n)^{\alpha n}] \leq n[\alpha - \beta(1-\exp(-\alpha/\beta))]. \quad (2)$$

Note that the inequalities are quite tight for reasonable n.

To approximate the behavior of the MHT, we can assume that the number of items placed in each sub-table exactly follows (1) (or, to give a little wiggle room, the near-tight inequality). Of course, these quantities may deviate somewhat from their expectations (particularly when αn is small), and so these are only heuristic approximations. Once the number of items under consideration is very small, one can use Markov's inequality; if the expected number of items in hashed into a sub-table is $z \ll 1$, then the probability that it is nonzero is at most z.

An exact calculation can be performed similarly. Here, we successively calculate the distribution of the number of items passed to the each sub-table, using the distribution from the previous sub-table. The computation uses the combinatorial fact that when r items are placed randomly into s buckets, the distribution of the number of buckets that remain empty can be calculated. For details, see Appendix III.

In Table I, we compare the results of the heuristic approximation with the exact calculations for a sample MHT. As one would expect, the approximation is extremely accurate when a reasonable number of items are hashed into a sub-table of reasonable size. However, as mentioned above, the approximation becomes less useful as the number of items left to be placed becomes small, since then the random variables of interest become less concentrated around their expectations.

[1]The original Broder–Karlin result shows that crises can be effectively dealt with in certain settings by replacing certain hash functions when a crisis occurs. Since rehashing is not a viable technique in our setting, we consider the occurrence of a crisis to be a serious failure, and so we need our high probability result to theoretically justify using the MHT.

TABLE I
THE APPROXIMATE AND EXACT EXPECTED NUMBERS OF ITEMS STORED IN EACH OF THE FIVE SUB-TABLES IN AN MHT FOR 10K ITEMS WITH $c_1 = 3$ AND $c_2 = 1/2$

	1	2	3	4	5
Approximate	8504.18	1423.70	71.78	0.34	-3.00×10^{-5}
Exact	8504.18	1423.67	71.80	0.35	1.62×10^{-5}

TABLE II
THE APPROXIMATE AND EXACT EXPECTED NUMBERS OF ITEMS STORED IN EACH OF THE FIVE SUB-TABLES IN AN MHT FOR 10K ITEMS WITH SUB-TABLE SIZES 40K, 10K, 5K, 2.5K, 2.5K (WHERE "K" DENOTES 1000)

	1	2	3	4	5
Approximate	8848.07	1088.11	63.42	0.40	-4.80×10^{-5}
Exact	8848.07	1088.08	63.45	0.41	3.37×10^{-5}

Table I also shows us that the distribution of the items in the MHT is highly skewed towards the first few tables. This property is extremely important, and it is a recurring theme in this work. Indeed, in Section VIII-B we show how to exploit this skew in our summaries, and we propose a slight modification to the original MHT design in order to create more skew to exploit. To illustrate that our approximation technique remains valid under this sort of modification, we compare the results of our heuristic approximation and our exact calculations for a sample modified MHT (which we discuss in more detail when we present more complete numerical results in Section IX) in Table II. As before, the approximation is quite accurate until the last sub-table.

B. The MHT Skew Property

We have mentioned that MHTs have a strong *skew property*, in the sense that the first sub-table contains most of the items, the second sub-table contains most of the rest, and so on. While this can be seen by experimenting with various values in (1), we provide a cleaner (albeit looser) presentation based on [3].

Suppose that we insert a set S_0 of items into the MHT, one-by-one. For $i = 1, \ldots, d$, let S_i be the set of items that are not placed in tables $T_1, \ldots, T_i$, and let m_i be the size of T_i. First, we note that $|S_i|$ is at most the number of pairwise collisions between elements of S_{i-1} in T_i. Next, we see that given S_{i-1}, there are $\binom{|S_{i-1}|}{2}$ possible pairwise collisions of elements in S_{i-1}, and each of these collisions occurs in T_i with probability $1/m_i$. By linearity of expectation,

$$\mathbf{E}[|S_i| \mid |S_{i-1}|] \leq \binom{|S_{i-1}|}{2} \cdot \frac{1}{m_i} \leq \frac{|S_{i-1}|^2}{2m_i}.$$

Using the heuristic approximation that $|S_i| \leq |S_{i-1}|^2/2m_i$, it is not difficult to show that the $|S_i|$'s decay doubly exponentially when $c_1 c_2 > 1/2$. Indeed, this result is the intuition for the choice of $d = O(\log \log n)$. The result also tells us to expect the distribution of the $|S_i|$'s to be very skewed.

VII. An Interpolation Search Summary

A straightforward approach to constructing a summary is to hash each item placed in the table to a uniformly distributed b-bit string, for sufficiently large b. We associate each such string with a value (requiring $\log d = \log\log\log n + O(1)$ bits) that indicates what hash function was used for the corresponding item. Searching for an item in the summary can now be done using interpolation search [11], which requires only $O(\log\log n)$ operations on average.

Insertions and deletions are conceptually easy; simply add or remove the appropriate string. However, interpolation search is typically defined over a sorted array of strings, and in this case, adding or removing a string in constant time requires a *block copy* operation in hardware to shift the contents of the array as needed. Using more sophisticated data structures, it is possible to implement the summary so that lookups, insertions, and deletions all require $O(\log\log n)$ time with high probability, where the insertion and deletion times are amortized. In certain applications, it may even be possible to strengthen the amortized bounds to worst-case bounds. For more details, see [8]. Such data structures result in a much more complicated summary than the Bloom filter-based constructions presented in Section VIII, especially if they must be implemented in hardware.

In this summary construction, a *failure* occurs if two items yield the same b bit string. In this case, one might not be able to record the proper value for each item. Therefore b must be chosen to be large enough to make this event extremely unlikely. Of course, b must also be chosen to be large enough so that the false positive probability is also quite small.

We note that two items with the same bit string do not actually constitute a problem if they hash to the same sub-table. For convenience we are choosing to call any such collision a failure here, since allowing any collisions would make handling deletions problematic. In principle, however, we could deal with such collisions if we so desired. One approach would be to record the maximum value associated with each string. In this case, when doing a search, one might have to look at multiple locations in the hash table. For example, if the value 2 is stored with the hash of an item, the item is most likely in the second sub-table, but it might be in the first. Since our goal is to guarantee that only one hash table lookup is necessary for each lookup operation, we do not consider this technique. However, this approach might be useful in some applications and similar ideas are applicable to our other summary constructions as well.

The failure and false positive probabilities for this summary can be computed very easily. The probability of a failure can be calculated using standard probabilistic techniques, as it is just a special case of the birthday paradox [18]. The probability of a false positive, conditioned on no failure occurring (so all n items have distinct b bit strings), is $n/2^b$.

For concreteness, we describe two specific instances of this scheme. Choosing $b = 61$ allows 3 bits for the associated value and for everything to fit into a 64 bit word; 3 bits is enough for 8 hash functions, which should be suitable for most implementations. Setting $n = 100\,000$ gives a failure probability less than 2.17×10^{-9} and a false positive probability (conditioned on no failure occurring) less than 4.34×10^{-14}. For $n = 10\,000$, we can achieve similar results for $b = 55$. For these values of n and b, the failure probability is less than 1.39×10^{-9} and the false positive probability (conditioned on no failure occurring) is 2.78×10^{-13}.

This scheme requires only $\Theta(n \log n)$ bits to ensure that failures occur with asymptotically vanishing probability; in this case, false positives occur with vanishing probability as well. In practice, however, the hidden constant factor is nontrivial, and hence the number of bits required can be significantly larger than for other approaches. Also, this approach requires that the application be amenable to a fast implementation of interpolation search. It is not clear whether any such application exists at this time, especially since we are not aware of any work on hardware implementations of interpolation search. Nevertheless, there are some theoretical advantages of this summary over the others discussed in this paper, most notably the ability to handle insertions and deletions easily (in certain cases) and the very small false positive probability.

VIII. Bloom Filter-Based MHT Summaries

In this section, we propose summaries that exploit the skew property of MHTs, making extensive use of the theory of Bloom filters. For now we consider insertions only, deferring our discussion of deletions until Section XI. We start with an initially empty summary and MHT and insert n items sequentially into both. The summaries presented here never require items to be moved in the MHT, and with high probability, they correctly identify the sub-tables storing each of the n items.

A. A Natural First Attempt

Our first Bloom filter-based MHT summary can be seen as a simple Bloomier filter that allows insertions. To better illustrate this point, we start by placing our problem in a general setting.

Suppose we have a set of n items, where each item has an integer *type* in the range $[1, \ldots, t]$. Our Bloom filter variant consists of m *cells*, where each cell contains a single value in $\{0, 1, \ldots, t\}$ (requiring $\log(t+1)$ bits), and k hash functions (whose domain is the universe of possible items). For convenience, we assume the m cells are divided into k disjoint groups of size m/k, and that each group is the codomain of one hash function. Alternatively, the structure could be built so that all k hash functions hash into the entire set of cells. This decision does not affect the asymptotics. However, the partitioned version is usually easier to implement in hardware, although the unpartitioned version may give a lower false positive probability [5].

Each cell in the structure initially has value 0. When an item is inserted, we hash it according to each of the hash functions to obtain the set of cells corresponding to it. For each of these cells, we replace its value with the maximum of its value and the type of the item. Thus, any cell corresponding to an inserted item gives an overestimate of the type of the item, and if some cell corresponding to an item has value 0, that item is not in the set represented by the structure. The lookup operation is now obvious; to perform a lookup for an item x, we hash x using the k hash functions and compute the minimum z of the resulting counters, and then either declare that x is not represented by the summary (if $z = 0$), or that x has type at most z (if $z > 0$). Note that the lookup operation may give several different kinds of errors: *false positives*, where the summary returns a positive type for an element not in represented set, and *type j failures*,

where the structure returns the incorrect type for an element of type j. The analysis of this structure now follows easily from the standard analysis of a Bloom filter [5].

Lemma 8.1: Suppose that we insert a set S of n items into the structure described above. Then the probability that a particular item $x \notin S$ gives a false positive is

$$(1 - (1 - k/m)^n)^k$$

and if there are $\beta_j n$ items of type greater than j, then the probability that a specific item of type j causes a failure is

$$(1 - (1 - k/m)^{\beta_j n})^k.$$

To use this structure as a summary for an MHT, we simply insert items into the structure as they are inserted into the MHT, and define the type of an item to be the sub-table of the MHT containing the item. (Of course, the type of an item is not well-defined if inserting it into the MHT causes a crisis; that is a different sort of failure that must be considered separately.) A false positive now corresponds to the case where the summary returns a positive type for an item not in the underlying MHT, and a type j failure now corresponds to the case where an item is in T_j in the underlying MHT, but the summary returns some other type when queried with that item. While false positives are not problematic if they appear sufficiently infrequently, we want to avoid any failures in our summary.[2]

In general, Lemma 8.1 can be used in conjunction with a union bound to bound the probability that there are any type j errors; if there are $\alpha_j n$ items of type j, then the probability that any type j failure occurs is at most

$$(\alpha_j n)(1 - (1 - k/m)^{\beta_j n})^k \approx (\alpha_j n)(1 - e^{-k\beta_j n/m})^k.$$

In our setting, Lemma 8.1 demonstrates that the most important tradeoff in constructing the summary is between the probability of a type 1 failure and the false positive probability, which both depend significantly on the numbers of hash functions used in the filter. Following the standard analysis from the theory of Bloom filters, to minimize type 1 failures, we would like $k = (\ln 2)m/\beta_1 n$. Typically this gives a rather large number of hash functions, which may not be suitable in practice. Further, this is far from the optimal number of hash functions to minimize false positives, which is $k = (\ln 2)m/n$, and therefore choosing such a large number of hash functions may make the false positive probability unreasonably high. In general, the choice of the number of hash functions must balance these two considerations appropriately.

There are other significant tradeoffs in structuring the MHT and the corresponding summary. Specifically, one can vary the number of sub-tables and their sizes in the MHT, as well as the size of the summary and the number of hash functions used. Generally, the more hash functions used in the MHT, the smaller the probability of a crisis (up to some point), but increasing the number of hash functions in the MHT increases the number of types, increasing the storage requirement of the summary structure. Moreover, the division of space in the MHT affects not only the crisis probability, but also the number of items of each type, which in turn affects the probability of failure.

As an aside, we note that several bit-level tricks can be used to minimize summary space. For example, three cells taking values in the range $[0, 5]$ can be packed into a single byte easily. Other similar techniques for saving bits can have a non-trivial impact on performance.

Asymptotically, choosing $m = \Theta(n \log n), k = \Theta(\log n)$, and using $\Theta(\log \log \log n)$ bits per cell suffices to have the probability of failure vanish, for a total of $\Theta(n(\log n) \log \log \log n)$ bits. The constant factors in this approach can be made quite small by taking advantage of skew. We present a complete analysis in Appendix II.

B. On Skew, and an Improved Construction

Lemma 8.1 highlights the importance of skew: the factor of β_j in the exponent drastically reduces the probability of a failure. Alternatively, the factor of β_j can be seen as reducing the space m required to achieve a certain false positive probability by a non-trivial constant factor.

In the construction above, the most likely failure is a type 1 failure; there are many fewer items of types greater than 1, and so there is very little probability for a failure for these items. A natural way to reduce the probability of a type 1 failure is to introduce more skew by making the size of the first sub-table larger (while still keeping linear total size). This can significantly reduce the number of elements of type larger than 1, shrinking β_1, which leads to dramatic decreases in the total failure probability (the probability that for some j, some item causes a type j failure). That is, if one is willing to give additional space to the MHT, it is usually most sensible to use it in the first sub-table. We use this idea when designing specific instances of our constructions in Section IX.

A problem with using a single filter for classifying items of all types is that we lose some control, as in the tradeoff between false positives and type 1 errors. Taking advantage of the skew, we suggest a *multiple Bloom filter* approach that allows more control and in fact uses less space, at the expense of more hashing. Instead of using cells that can take on any of $t + 1$ values, and hence requiring roughly $\log_2(t + 1)$ bits to represent, our new summary consists of multiple Bloom filters, $B_0, B_1, \ldots, B_{t-1}$. The first Bloom filter, B_0, is simply used to determine whether or not an element is in the MHT; that is, it is a standard, classical Bloom filter for the set of items in the MHT. In convenient terms, it separates items of type greater than or equal to 1 from elements not in the MHT, up to some small false positive probability. (But note that if an element gives a false positive in B_0, we do not care about the subsequent result.) Next, B_1 is a standard Bloom filter designed to represent the set of items with type greater than or equal to 2. An item that passes B_0 but not B_1 is assumed to be of type 1 (and therefore in the first sub-table of the MHT). A false positive for B_1 on an item of type 1 therefore leads to a type 1 failure, and hence we require an *extremely small* false positive probability for the filter to avoid such a failure. We continue on with $B_2, B_3, \ldots, B_{t-1}$ in the corresponding way (so the assumed type of an item x that passes B_0 is the smallest j such that x does not pass B_j, or t if x passes all of $B_0, B_1, \ldots, B_{t-1}$).

[2]Technically, we may wish to differentiate between the false positive probability and the false positive rate, as defined in [12], but the distinction is unimportant in practice. See [12] for an explanation.

Because of the skew, each successive filter can be smaller than the previous one without compromising the total failure probability. The skew is key for this approach to yield a suitably small overall size. Indeed, the total size using multiple Bloom filters will often be less than using a single filter as described in Section VIII-A; we provide an example in Section IX. Further, by separating the filters, one can control the false positive probability and the probability of each type of error quite precisely. Also, by separating each type in this way, at some levels small Bloom filters could be replaced by lists of items, for example using a Content Addressable Memory (CAM).

The only downside of this approach is that it can require significantly more hashing than the others. For many applications, especially ones where all of the hashing computation is parallelized in hardware, this may not be a bottleneck. Also, it turns out that the calculations in Section IX hold even if the hash functions used by the B_i's are not independent, as long as all of the hash functions used in any particular B_i are independent. Thus, if ℓ is the least common multiple of the sizes of the codomains of the hash functions, we can just use a few hash functions with codomain $\{0,\ldots,\ell-1\}$, and compute hashes for the B_i's by evaluating those hash functions modulo the sizes of the desired codomains.

As a further improvement, one might try to combine this technique with a variant of double hashing [9], [12] to reduce the total number of hash functions even further. However, despite the encouraging practical and asymptotic results surrounding double hashing, some simple experiments suggest that this approach is not effective in our setting because we require extremely small false positive probabilities. The situation appears to improve as we increase the number of hash functions (triple hashing, quadruple hashing, etc.), but our desire for extremely small false positive probabilities makes it impossible to prove the effectiveness of this modification through experiments. If this approach could somehow be proven viable, however, it would allow for a drastic reduction in the required amount of hashing computation, although it would not reduce the number of bits of the summary that must be examined for each lookup operation.

As for asymptotics, if we knew that for each $j=1,\ldots,d-1$, there were at most $X_{\geq j}$ items of type at least j, then $\Theta(X_{\geq j}\log n)$ bits in B_j would suffice for the probability of a type j failure to vanish. Since $X_{\geq j} \leq n$ and $d=O(\log\log n)$, the failure probability can be made to vanish with $O(n(\log n)\log\log n)$ bits. However, it turns out that since the first $\log\log n+\Theta(1)$ of the $X_{\geq j}$'s decay doubly exponentially with high probability for a well-designed MHT, we can actually get the failure probability to vanish with $\Theta(n\log n)$ bits. We give the proof in Appendix II.

IX. Numerical Evaluation (Insertions Only)

In this section, we present constructions of our three summaries for 10 000 and 100 000 items and compare their various storage requirements, false positive probabilities, and failure probabilities. For completeness, we compare with results from Song *et al.* [20]. We continue to work in the setting where there are insertions, but not deletions, because handling deletions introduces too many subtle issues to allow for a straightforward comparison. Nevertheless, this restriction allows for a fair comparison against the scheme in [20], which requires additional structures to handle deletions.

For the MHT summaries, our preliminary goal is to use at most 6 buckets per item; this is less than 1/2 the size of the hash table (in terms of buckets) in [20], and seems like a reasonable objective. From here, we arrive at the following underlying MHTs (where "k" represents 1000). For 10k items, there are five sub-tables, with sizes 40k, 10k, 5k, 2.5k, and 2.5k, giving a crisis probability less than 1.01×10^{-12} (calculated using the method of Section VI-A). For 100k items, there are 6 sub-tables, with sizes 400k, 100k, 50k, 25k, 12.5k, and 12.5k, giving a crisis probability less than 7.78×10^{-16}. Both of these crisis probabilities are dominated by the failure probabilities of the corresponding summaries (except in one case, with 10k items using multiple filters, where the crisis probability is still smaller than the failure probability).

Also, for the MHT summaries discussed in Section VIII, we do not attempt to optimize all of the various parameters. Instead we simply exhibit parameters that simultaneously perform well with respect to all of the metrics that we consider. Also, we note that it is not practical to exactly compute the false positive and failure probabilities for these schemes. However, it is possible to efficiently compute estimates of these probabilities, and we have built a calculator for this purpose. We believe that our estimates are fairly tight upper bounds when the probabilities are very small, and so we use them as if they were the actual probabilities. For more details, see Appendix III.

We configure the Bloom filter-based MHT summaries as follows. For 10k items, we configure our first summary to have 120k cells and 15 hash functions. When computing the storage requirement, we assume that 3 cells (each taking integral values in [0,5]) are packed into a byte. For the multiple Bloom filter summary, we use filters of sizes 106k, 87.5k, 5.5k, 500, and 100 bits, with seven hash functions for the first filter and 49 for each of the others.[3] For 100k items, we configure the first Bloom filter based summary to have 1.2m cells (where "m" represents one million) and 15 hash functions, and here we use three bits for each cell (taking integral values in [0,6]). We configure the multiple Bloom filter summary to have filters of sizes 1.06m, 875k, 550k, 1k, and 1k, with 7 hash functions for the first filter and 49 for each of the others.

The results of our calculations are given in Table III. In that table, IS denotes the interpolation search scheme of Section VII, SF denotes the single filter scheme of Section VIII-A, and MBF denotes the multiple Bloom filter scheme of Section VIII-B. We configure the interpolation search summary according to the examples in Section VII. The notation "*" for the Song *et al.* [20] summary denotes information not available in that work. All storage requirements for our summaries include the space for the bit table mentioned in Section V.

In the last column of Table III, note that the sum of the failure and crisis probabilities can be thought of as a bound on the overall probability that a scheme does not work properly. (Also, as mentioned previously, except in the case of 10k items with multiple filters, the failure probability dominates). As can be seen in the table, interpolation search performs extremely well at

[3]For the multiple Bloom filter construction, we use unpartitioned Bloom filters, so the number of hash functions need not divide the size of a filter.

TABLE III
NUMERICAL RESULTS

(a) 10k items

Scheme	Hash Table Size (buckets)	Summary Space (bytes)	False Positive Probability	Failure + Crisis Probability
[20]	131072	49152	.002	*
IS	60000	80000	2.78×10^{-13}	1.39×10^{-9}
SF	60000	47500	.006	7.64×10^{-10}
MBF	60000	32450	.006	4.97×10^{-12}

(b) 100k items

Scheme	Hash Table Size (buckets)	Summary Space (bytes)	False Positive Probability	Failure + Crisis Probability
[20]	*	*	*	*
IS	600000	875000	4.34×10^{-14}	2.17×10^{-9}
SF	600000	525000	.006	7.27×10^{-9}
MBF	600000	379000	.006	1.38×10^{-11}

the expense of a fairly large summary. The single filter scheme appears comparable to the structure of [20] for 10k items, but uses much less space. The multiple filter scheme allows further space gains with just slightly more complexity. Our schemes also appear quite scalable; for 100k items, we can maintain a ratio of 6 buckets per item in the hash table, with just a slightly superlinear increase in the summary space for our proposed schemes.

For the scheme presented by Song *et al.*, there is no failure probability as we have described, as an item will always be in the location given by the summary. There may, however, be a crisis, in that some bucket may have more than one item. (Technically, there can be a failure, because they use only three-bit counters with ten hash functions; however, the probability of a failure is very, very small and can be ignored.) They do not have any numerical results for the crisis probability of their scheme when including their heuristics, and hence we leave a "*" in our table of results. However, they do report having found no crisis in one million trials. Finally, we note that [20] does not include results for 100k items. Since it is not clear how to properly configure that scheme for 100k items, we do not attempt to analyze it.

It is worth noting that our improvements in summary space over the scheme in [20] are not as dramatic as the improvement in hash table size. The intuitive explanation for this phenomenon is that the hash table in [20] seems to require $\Theta(n \log n)$ buckets in the hash table, while the MHT requires only $\Theta(n)$ buckets, giving our schemes a factor $\Theta(\log n)$ reduction in the size of the hash table. However, from Sections IV, VII, VIII-A, and VIII-B, we know that if we require the failure probability to vanish, the summary in [20] seems to require $\Theta(n(\log n)\log\log n)$ bits, the interpolation search summary requires $\Theta(n \log n)$ bits, the single filter summary requires $\Theta(n(\log n)\log\log\log n)$ bits, and the multiple Bloom filter summary requires $\Theta(n \log n)$ bits. Thus, while these summaries seem to require fewer bits than the one in [20], the gain appears to be smaller than the factor $\Theta(\log n)$ reduction in hash table size.

X. EXPERIMENTAL VALIDATION (INSERTIONS ONLY)

Ideally, we would be able to directly verify the extremely low failure probabilities given in the previous section through experiments. However, since the probabilities are so small, it is impractical to simulate the construction of the summaries sufficiently many times to accurately estimate the real failure probabilities. We have attempted to validate the calculator we have developed for the summaries based on Bloom filters, and have found that it does give an upper bound in all of our tests. In fact it can be a fairly weak upper bound when the failure probability is very large (greater than 0.1, for example). In all our experiments, we simulated random hashing by fixing hashes for each item using a standard 48-bit pseudorandom number generator.

We have simulated the single filter for 10k items in Table III; in one million simulations, we saw no errors or crises, as predicted by our calculations. We also experimented with a variant on this filter with only 100k counters and 10 hash functions. Our calculations for this filter gave an upper bound on the probability of failure of just over 2.1×10^{-6}; in one million trials, we had one failure, a natural result given our calculations.

While further large-scale experiments are needed, our experiments thus far have validated our numerical results.

XI. DELETIONS

Handling deletions is substantially more difficult than handling insertions. For example, the scheme proposed in [20] for handling deletions requires significant memory; it essentially requires a separate version of the hash table that records all of the hash locations of every item. Moreover, deletions can require significant repositioning of elements in the hash table. To address these issues, we explore two deletion paradigms: lazy deletions and counter-based deletion schemes.

A. Lazy Deletions

A natural, general approach is to use *lazy* deletions. That is, we keep a *deletion bit* array with one bit for each cell in the hash table, initially 0. When an item is deleted from some bucket b, we simply set the deletion bit corresponding to b to 1. When looking up an item, we treat it as deleted if we find it in a bucket whose deletion bit is 1. When a preset number of deletions occurs, when the total number of items in the hash table reaches some threshold, or after a preset amount of time, we can reconstruct the entire data structure (that is, the hash table, the deletion bit array, and the summary) from scratch using the items in the hash table, leaving out the deleted items. If we want to guarantee good performance whenever there are at most αn deleted items and at most n undeleted items in the hash table, it suffices to simply build our data structures to be able to cope with $(1+\alpha)n$ items, rebuilding them whenever there are at least $(1+\alpha)n$ items in the hash table, at least αn of which are marked for deletion. The obvious disadvantage of this approach is that expensive reconstruction operations are necessary, potentially frequently, depending on how often insertions occur. Also, extra space is required to maintain the deleted items until this reconstruction occurs.

However, reconstruction operations are much cheaper than one might expect. Indeed, the time required to perform the MHT reconstruction is essentially determined by the number of items in the MHT that must be moved, and the time required to perform the summary reconstruction is essentially just the time required to scan the MHT and rehash all of the items using the summary hash functions. We find that the MHT skew property ensures that very few items need to be moved in the MHT during

a reconstruction, which tells us that the MHT reconstruction is much less expensive than the analogous procedures for other multiple-choice hash tables.

As for the scan of the MHT required to rebuild the summary, we note that while this procedure may be somewhat expensive, its simplicity may be an asset, and it may be cheaper than moving the items in the MHT. Also, the cost of this scan can be ameliorated in various ways. For example, if we store the summary hash values of the items in a structure analogous to the MHT (in slow memory), then we only need to scan the hash values of the items when reconstructing the summary, as opposed to scanning the items themselves, which might be much larger. Using the hash reduction techniques discussed in Section VIII-B reduces the storage requirement of this data structure, further reducing the cost of the scan.

We now focus on the number of items that must be moved during a reconstruction of the MHT. We consider a natural implementation of the MHT rebuilding process. Before proceeding, recall that the MHT consists of tables $T_1, \ldots, T_d$ and corresponding hash functions $h_1, \ldots, h_d$, and that an item x should be placed in $T_j[h_j(x)]$, where j is as small as possible subject to $T_j[h_j(x)]$ being unoccupied. The natural MHT reconstruction algorithm is then as follows: for $i = 1, \ldots, d$, iterate over the items in T_i (using the bucket occupancy bit table described in Section V), and for each item x in T_i, determine (again using the bucket occupancy table) if there is some smallest $j < i$ such that $T_j[h_j(x)]$ is empty, and move x to $T_j[h_j(x)]$ if this is the case. (Of course, a bucket is considered empty if its deletion bit is set; we move an item by copying it to its destination and then marking its origin as deleted; we update the occupancy bit table as we go; and at the end of the algorithm, we reset all of the deletion bits to 0.)

Consider, for example, the case where we reconstruct an MHT with $(1 + \alpha)n$ items, exactly αn of which are marked as deleted. For the moment, assume that the deleted items are chosen randomly. While this assumption may be unrealistic, it allows us to analyze the number M of items moved and gives a good indication of general performance. To this end, we let M' be the number of items x that are not deleted and are in some table $j > 1$ such that at least one of the items in $T_1[h_1(x)], \ldots, T_{j-1}[h_{j-1}(x)]$ is deleted. We claim that $\mathbf{E}[M']$ can be used as an approximate upper bound on $\mathbf{E}[M]$. Indeed, if, for example, an item y in T_1 is deleted, it is intuitively likely that there are a few items x with $h_1(x) = h_1(y)$, but unlikely that all of those items are moved. However, those items x that are not moved are counted in M' but not M, which suggests that $\mathbf{E}[M']$ is an approximate upper bound for $\mathbf{E}[M]$. The only complication is that an item x in T_j for some $j > 2$ can be moved even if none of $T_1[h_1(x)], \ldots, T_{j-1}[h_{j-1}(x)]$ are deleted; one of those items could simply move. But in practice we expect that almost all moves will be from items in the second sub-table to buckets in the first sub-table, and so we expect the effect of this complication to be minimal.

To estimate $\mathbf{E}[M']$, we consider some item x in table $j > 1$. The probability that x is not deleted is clearly

$$\prod_{i=0}^{\alpha n-1} \left(1 - \frac{1}{(1+\alpha)n - i}\right)$$

and given that x is not deleted, the probability that there is some $i < j$ such that the item at $T_i[h_i(x)]$ is deleted is

$$1 - \frac{\binom{(1+\alpha)n-j}{\alpha n}}{\binom{(1+\alpha)n-1}{\alpha n}} = 1 - \prod_{i=1}^{j-1} \frac{1}{1 + \frac{\alpha n}{n-i}}.$$

From these probabilities and the expected numbers of items in sub-tables when $(1 + \alpha)n$ items are inserted (which are obtainable using the method of Section VI-A), we can easily compute $\mathbf{E}[M']$. Asymptotic high probability bounds can then be obtained by standard martingale techniques (for example, [18, Sec. 12.5]).

We give a concrete example using a sample MHT for 10 000 items discussed in Section IX. The MHT consists of five sub-tables, with sizes 40k, 10k, 5k, 2.5k, and 2.5k, respectively (where "k" denotes 1000). We set $\alpha = 0.1$ and $n = 9090$, so that $(1+\alpha)n < 10\,000$. Performing the calculations above, we find that $\mathbf{E}[M'] \approx 100.02$. That is, only about 1.1% of the n items are moved on average during an MHT reconstruction! This result shows that periodic MHT reconstructions in the standard lazy deletion scheme are likely to be significantly less expensive than full reconstructions.

To confirm the correctness of our calculations, we estimated the expected number of moves required by an MHT reconstruction in a simple experiment. We averaged the required number of moves over 100 000 trials, where each trial consisted of inserting $(1 + \alpha)n$ items into an initially empty MHT, deleting αn of those items at random, and then counting the number of moves required to reconstruct the MHT. The resulting estimate of $\mathbf{E}[M]$ was 96.98, which is smaller than but close to the calculated value, as expected. The minimum and maximum observed values of M were 58 and 113, respectively, demonstrating that M is unlikely to deviate too far above $\mathbf{E}[M']$.

While it may be initially surprising that reconstructing the MHT requires so few moves, there is some simple intuition behind the result. Indeed, Table II suggests that (approximately) 88% of the $(1 + \alpha)n = 1.1n$ items are placed in the first sub-table and the rest are placed in the second sub-table. Under this assumption, an item is moved only if it is initially placed in the second sub-table, it is not deleted, and the item at its hash location in the first sub-table is deleted. But only 12% of the items are initially placed in the second sub-table, and only about $1 - \alpha = 90\%$ of them are not deleted, and of those that remain, only about $\alpha = 10\%$ hash to buckets in the first sub-table that contain deleted items. Thus, the fraction of the n items that are moved is about $(1.1)(0.12)(0.9)(0.1) \approx 1.2\%$, closely matching the results above.

One might reasonably wonder how dependent these results are on our assumption that deletions occur randomly. As evidence that the results are fairly robust, we now consider a more pessimistic deletion model. As before, we assume that we perform a reconstruction of an MHT with $(1 + \alpha)n$ items, exactly αn of which are marked as deleted. However, rather than assuming that the αn deleted items are chosen randomly from all $(1+\alpha)n$ items, we assume that they are chosen randomly from the items in the first sub-table of the MHT. (For a well-designed MHT, it is overwhelming likely that there are at least αn items

in the first sub-table.) Then, as before, we focus on the number M of items that must be moved.

To analyze $\mathbf{E}[M]$, we let S_1 denote the set of items not placed in first sub-table. We let M' be as before, and expect that $\mathbf{E}[M']$ is an approximate upper bound on $\mathbf{E}[M]$. Clearly, if $|S_1| > n$, then $M' = n$. Next, consider the case where $|S_1| \leq n$. Then any particular item x of any type $j > 1$ is counted in M' if and only if the item at $T_1[h_1(x)]$ is deleted, which occurs with probability

$$1 - \prod_{i=0}^{\alpha n - 1} \left(1 - \frac{1}{(1+\alpha)n - |S_1| - i}\right).$$

By linearity of expectation, we conclude that

$$\mathbf{E}[M' \mid |S_1|] = \begin{cases} n, & \text{if } |S_1| > n \\ |S_1|\left(1 - \prod_{i=0}^{\alpha n-1}\left(1 - \frac{1}{(1+\alpha)n-|S_1|-i}\right)\right), & \text{otherwise} \end{cases}$$

which is easily computed. We can then compute the distribution of $|S_1|$ (using the method of Section VI-A) and use it to calculate $\mathbf{E}[M']$.

We examined this deletion model for the same MHT and parameters as before. We calculated that $\mathbf{E}[M'] \approx 118.34$, which is approximately 1.3% of the n items. Through an experiment (with 100 000 trials, as before), we also estimated $\mathbf{E}[M]$ as about 114.25. Over the course of the experiment, the smallest and largest observed values of M were 70 and 166, respectively. We concluded that M is reasonably concentrated around $\mathbf{E}[M]$.

We can also predict the above results by slightly modifying the intuitive reasoning for the original deletion model. As before, about 8848 of the $(1+\alpha)n = 1.1n = 9999$ items are placed in the first sub-table. The probability that any particular item x of the remaining 1151 items must be moved is therefore about $\alpha n/8848 \approx .1$. Therefore, the fraction of the n items that must be moved is about $(1.1)(1151/9999)(0.1) \approx 1.3\%$ (again neglecting the fact that only one item in the second sub-table can be moved into a particular bucket in the first sub-table).

One might hope to improve these results by allowing items marked for deletion in the MHT to be overwritten by newly inserted items. We do not provide a detailed analysis of this approach, but do give some basic caveats. First, we expect any gains to be minor, given the excellent performance of the basic lazy deletion scheme. Second, one must beware of additional pollution in the summary. In the specific case of our Bloom filter-based summaries, when an item x that is marked for deletion is overwritten by a new item, x cannot be simply removed from the summary (just as a standard Bloom filter does not support deletions). But leaving x in the summary while adding the new item adds noise to the summary, increasing its false positive rate and failure probability. Furthermore, all future queries for x will now result in false positives. In situations where a recently deleted item is likely to be the subject of a lookup query, the latter issue may constitute a very serious problem. Of course, both summary pollution issues could be ameliorated by increasing the size of the summary and/or adding additional data structures (e.g., to track overwritten items), but such modifications add non-trivial overhead and complications. Therefore, we believe that for most applications, the best approach is likely to be either the standard lazy deletion scheme analyzed above or one of the counter-based schemes that we discuss next in Section XI-B.

B. Counter-Based Deletion Schemes

While the lazy deletions schemes of Section XI-A may be appropriate for many applications, their reliance on periodic reconstructions of the summary might preclude their use in certain situations, such as very high speed data streams. In other words, one might require good worst case time bounds for hash table operations and be unable to settle for the amortized bounds offered by lazy deletion schemes. Of course, in order to achieve good worst case bounds, the underlying hash table must allow for easy deletions. Indeed, an item can be deleted from an MHT simply by removing it from its bucket; no repositioning of the elements is necessary. This observation is another excellent reason for using the MHT as our underlying hash table.

Now, whenever an item is deleted from its bucket in the MHT, our summary must be updated to reflect the deletion. The Bloom filter-based summaries of Section VIII can be easily modified so that these updates can be performed quickly. For the single filter summary of Section VIII-A, we require that each cell now contain one counter for each type, and that each counter tracks the number of items in the MHT of the corresponding type that hash to the cell containing the counter. Both insertions and deletions can now be performed extremely quickly, simply by incrementing or decrementing the appropriate counters. A similar modification works for the multiple Bloom filter summary of Section VIII-A; we simply replace each Bloom filter by a counting Bloom filter.

Unfortunately, these modified summaries consume much more space than the originals. However, the space requirements of the modified Bloom filter-based summaries can be minimized by aggressively limiting the number of bits used for each counter. Of course, we must guarantee that the probability that some counter overflows is extremely small, since the existence of an overflow can eventually lead to the summary returning an incorrect answer.

Choosing the appropriate number of bits for a counter therefore requires some work. First, we observe that if there are n elements associated with m counters (all initially 0), and for each element we increment c randomly chosen counters, then the distribution of the resulting maximum counter value is the same as the distribution of the maximum load resulting from throwing nc balls randomly into m bins. If nc/m is not too small, we can derive nontrivial high probability bounds for this distribution using the Poisson approximation [18, Sec. 5.4]. For the Bloom filter-based summaries, this approach allows us to keep the sizes of the counters reasonable while simultaneously ensuring that, with high probability, no false negatives or type j failures occur, for all but the largest couple values of j. This approach is effective until the expected number of items in a table becomes so small that either the variance is too big or the Poisson approximation is inaccurate.

One approach might be to avoid the problems caused by small sub-tables by replacing them with a CAM. Alternatively, if we

use small sub-tables, the expected number of items that hash to any counter in them is so small each counter can be represented by a bit or two. A detail that must be dealt with is that for a small table, if multiple hash functions for an item being inserted (or deleted) hash to the same counter, that counter should be incremented (or decremented) only once. Otherwise, the insertion of a single item could, with small but non-negligible probability, result in a particular counter being incremented multiple times. This issue is not problematic when we have larger counters and hash tables, and it does introduce some overhead, so we only modify the increment and decrement operations in this way for smaller tables, which are rarely used (by the MHT skew property).

As a concrete example of our design techniques, we modify the multiple bloom filter summary for 10 000 items given in Section IX to use counting Bloom filters with appropriately sized counters. Based on the heuristic calculations described above, we suggest using the modified insertion and deletion operations in the last two filters, using 4 bits per counter in the first three filters, 3 or 4 bits per counter in the fourth filter, and 1 or 2 bits per counter in the last filter. We suspect that the more conservative choices might be necessary to obtain failure probabilities comparable to those in Section IX, but that the less conservative ones are still fairly effective.

The two choices give summaries of essentially the same size. For the conservative choices, we can easily build the summary so that it uses 99 775 bytes (with two 4-bit counters per byte in the first four filters, and four 2-bit counters in the last filter), neglecting the 7500 bytes needed for the bit table described in Section V. The less conservative choices yield a summary that uses at least 99 700 bytes (neglecting byte-packing issues for the 3-bit counters). In both cases, the total storage requirement of the summary (including the 7500 byte bit table) is essentially 3.3 times that of the corresponding summary in Section IX.

To test the resulting data structure, we instantiated the summary (with 16-bit counters) and the underlying MHT with 10 000 items one million times and recorded the largest counter values $v_0, \ldots, v_4$ ever seen in each of the filters $B_0, \ldots, B_4$. The results were $v_0 = 12, v_1 = 11, v_2 = 12, v_3 = 4$, and $v_4 = 1$. We concluded that the results of the experiment were consistent with the results of our heuristic analysis. More detailed analysis and more extensive simulations could provide more insight into appropriate values for extremely small failure probabilities comparable to those in Section IX.

XII. Conclusions and Further Work

We have shown that designing small, efficient summaries to use in conjunction with multiple-choice hashing schemes is feasible, improving on the results of [20]. We believe the fact that our summaries can be analyzed to bound performance is a useful characteristic that will ease adoption.

There are several potential extensions to this work. Our ideas and theoretical results can be extended easily to the case where buckets can hold any constant number of items, but more analysis and experiments must be done. Also, more experimentation could be done to test our summary structures, including large-scale tests with hash functions commonly used in practice, as well as tests for specific applications. A detailed analysis of deletion workloads to determine the effect of and best approach for deletions would also be worthwhile.

Appendix I
An Asymptotic Bound on the Crisis Probability

This Appendix is devoted to the following result, which we consider to be the theoretical justification for the MHT's extremely low crisis probabilities.

Theorem 1.1: Suppose we hash n items into an MHT with tables $T_1, \ldots, T_d$ (with corresponding fully random hash functions $h_1, \ldots, h_d$), where the size of T_i is $m_i' = \lceil m_i \rceil$ for $m_i = c_2^{i-1} c_1 n$, for any constants $c_1 > 1$ and $c_2 < 1$ with $c_1 c_2 > 1$. Then for any constant $c > 0$, we can choose $d = \log\log n + \Theta(1)$ so that the probability that a crisis occurs is $o(n^{-c})$.

Proof: We begin the proof with a sequence of lemmas.

Lemma 1.1: Let S_0 denote the set of items being hashed, and for $i = 1, \ldots, d$, let S_i denote the set of items not placed in the first i tables. Then for $i \geq 1$ and any $B \geq |S_{i-1}|^2/m_i$, we have $\mathbf{E}[|S_i| \,|\, |S_{i-1}|] \leq |S_{i-1}|^2/2m_i$ and $\mathbf{Pr}(|S_i| > B \,|\, |S_{i-1}|) < (e/4)^{B/2}$.

Proof: Condition on $S_{i-1} = \{x_1, \ldots, x_\ell\}$. For $k = 1, \ldots, \ell$, let Y_k indicate whether there exists $j < k$ with $h_i(x_j) = h_i(x_k)$, so that $|S_i| = \sum_k Y_k$. Also, let $Z_1, \ldots, Z_\ell$ be independent 0/1 random variables with $\mathbf{E}[Z_k] = \min(1, (k-1)/m_i)$, let $Z = \sum_k Z_k$, and note that $\mathbf{E}[Z] \leq |S_{i-1}|^2/2m_i \leq B/2$. It is easy to see that

$$\mathbf{Pr}(Y_k = 1 \,|\, \{h_i(x_j) : j < k\}) = |\{h_i(x_j) : j < k\}|/m_i' \leq \mathbf{E}[Z_k].$$

It follows that $\mathbf{Pr}(Y_k = 1 \,|\, Y_1, \ldots, Y_{k-1}) \leq \mathbf{E}[Z_k]$. Thus,

$$\mathbf{E}[Y_{j_1} \cdots Y_{j_r}] \leq \mathbf{E}[Z_{j_1} \cdots Z_{j_r}]$$

for any $j_1, \ldots, j_r$, implying that $\mathbf{E}[|S_{i-1}|^j] \leq \mathbf{E}[Z^j]$ for any integer $j \geq 0$. Now for any $t > 0$,

$$\mathbf{E}[e^{t|S_i|}] = \sum_{j=0}^{\infty} \frac{t^j \mathbf{E}[|S_i|^j]}{j!} \leq \sum_{j=0}^{\infty} \frac{t^j \mathbf{E}[Z^j]}{j!} = \mathbf{E}[e^{tZ}].$$

We can now complete the proof by deriving a Chernoff bound in the usual way (e.g., [18, Theorem 4.4]). ∎

Lemma 1.2: Let $\{z_i\}_{i \geq 0}$ be a sequence where $z_0 = n$ and $z_i = z_{i-1}^2/m_i$ for $i \geq 1$. Then for $i \geq 0$, we have $z_i = (1/c_1 c_2)^{2^i - 1} c_2^i n$.

Proof: The proof is an easy induction on $i \geq 0$. ∎

Lemma 1.3: For any events $A_1, \ldots, A_\ell$,

$$\mathbf{Pr}\left(\bigcup_i A_i\right) \leq \sum_i \mathbf{Pr}\left(A_i \,\middle|\, \bigcap_{j<i} \neg A_j\right).$$

Proof: This result is standard, and the proof is trivial. ∎

Lemma 1.3: For any $i \geq 1$, we have

$$\mathbf{Pr}(\exists j \leq i : |S_j| > z_j) \leq i(e/4)^{z_i/2}.$$

Proof: Applying Lemmas 1.3, 1.1, and 1.2 gives

$$\begin{aligned}\mathbf{Pr}(&\exists j \le i : |S_j| > z_j) \\ &\le \sum_{j=1}^{i} \mathbf{Pr}(|S_j| > z_j \,||S_{j-1}| \le z_{j-1}) \\ &\le \sum_{j=1}^{i} (e/4)^{z_j/2} \le i(e/4)^{z_i/2}.\end{aligned}$$

■

We are now ready to prove the theorem. By Lemma 1.2, we can choose $r = \log\log n + \Theta(1)$ and obtain

$$2(c+1)\log_{4/e} n \le z_r = O(\sqrt{n}),$$

so Lemma 1.4 implies that

$$\mathbf{Pr}(|S_r| > z_r) < rn^{-c-1} = o(n^{-c}).$$

Since $z_r^2/m_{r+1} = O(\log n)$, Lemma 1.1 tells us that we can choose some $w = O(\log n)$ so that

$$\mathbf{Pr}\left(|S_{r+1}| > w \mid |S_r| \le z_r\right) \le n^{-(c+1)} = o(n^{-c}).$$

Markov's inequality and Lemma 1.1 now imply that for $i \ge 1$ and any $w' \le w$

$$\begin{aligned}\mathbf{Pr}(&|S_{r+1+i}| \ge 1 \mid |S_{r+i}| = w') \\ &\le \mathbf{E}[|S_{r+1+i}| \mid |S_{r+i}| = w'] \\ &\le w^2/2m_{r+1+i} = O\left(\frac{\log n}{n}\right).\end{aligned}$$

Since $|S_0| \ge |S_1| \ge \cdots \ge |S_d|$,

$$\begin{aligned}\mathbf{Pr}(&|S_{r+1+\lceil c+1\rceil}| \ge 1 \mid |S_{r+1}| \le w) \\ &\le \prod_{i=1}^{\lceil c+1\rceil} \mathbf{Pr}(|S_{r+i+1}| > 1 \mid |S_{r+i}| \le w) \\ &\le O\left(\frac{\log n}{n}\right)^{\lceil c+1\rceil} = o(n^{-c}).\end{aligned}$$

Setting $d = r + 1 + \lceil c+1\rceil$ and applying Lemma 1.3 gives

$$\begin{aligned}\mathbf{Pr}(|S_d| \ge 1) \le\ &\mathbf{Pr}(|S_r| > z_r) \\ &+ \mathbf{Pr}(|S_{r+1}| > w \mid |S_r| \le z_r) \\ &+ \mathbf{Pr}(|S_d| \ge 1 \mid |S_{r+1}| \le w) \\ =\ &o(n^{-c})\end{aligned}$$

completing the proof.

Appendix II
Asymptotics of the Bloom Filter Summaries

This Appendix provides rigorous analyses of the asymptotics for the Bloom filter-based summaries of Section VIII.

A. The Single Filter Summary

We show that for any constant $c > 0$, it is possible to construct the single filter summary of Section VIII-A so that it has failure probability $o(n^{-c})$ and requires $m = O(n\log n)$ bits and $k = O(\log n)$ hash functions. We assume that the filter is partitioned into k sub-filters of size m/k, one for each hash function, although our analysis can be modified for an unpartitioned filter.

Proceeding, we set $c' = (c+2)/\ln 2$, and then choose the smallest $m \ge c' n \log_{10/7} n$ such that for $k = \lfloor (m/n)\ln 2\rfloor$, we have that k divides m. The probability that a particular item gives a failure, regardless of its type, is then at most

$$\begin{aligned}(1-(1-k/m)^n)^k &\le \left(1 - e^{-nk/m - nk^2/m^2}\right)^k \\ &\le \left(1 - e^{-\ln 2 - (\ln 2)^2/n}\right)^k \\ &\le (0.7)^k \le (10/7)n^{-c-2}\end{aligned}$$

where the first step follows from the inequality $1 - x \ge e^{-x-x^2}$ for $x < 1/2$. Taking a union bound over all n items yields a total failure probability of $o(n^{-c})$.

B. The Multiple Bloom Filter Summary

We now show that for the MHT of Theorem 1.1, it is possible to construct a multiple Bloom filter summary with failure probability $o(n^{-c})$ that uses $O(n\log n)$ bits, where $c > 0$ is the same constant as in Theorem 1.1.

We continue to use the notation introduced in the statement and proof of Theorem 1.1. As in Section VIII-B, let $B_0, \ldots, B_{d-1}$ denote the Bloom filters in the summary. Let b_j denote the size of B_j, and let k_j denote the number of hash functions used by B_j. For simplicity, we assume that each B_j is partitioned into k_j sub-filters of size b_j/k_j, one for each hash function, although our analysis can be modified for unpartitioned filters.

By the same argument as in Appendix II.A, conditioned on there being at most y_{j-1} elements of type at least j, if $b_j \ge c' y_{j-1}\log_{10/7} n$ for $c' = (c+2)/\ln 2$ and $k_j = \lfloor (b_j/n)\ln 2\rfloor$, then the probability that any particular item of type j yields a type j failure is at most $(10/7)n^{-(c+2)}$. Taking a union bound over all n items gives an overall failure probability of $o(n^{-c})$.

To complete the proof, it suffices to show that we can choose the y_j's so that $\sum_{j=0}^{d-1} y_j = O(n)$ and $\mathbf{Pr}(\exists j : |S_j| > y_j) = o(n^{-c})$. To this end, we set $y_0 = n$, followed by $y_j = (1/c_1c_2)^{2^j-1}n$ for $j = 1, \ldots, r$, and then $y_j = y_r$ for $j = r+1, \ldots, d-1$. Then

$$\begin{aligned}\sum_{j=0}^{d-1} y_j &\le (d-r)n + \sum_{j=1}^{r} y_j \\ &\le (d-r)n + n\sum_{j=1}^{\infty} (1/c_1c_2)^{2^j-1} = O(n)\end{aligned}$$

where we have used the fact that $d - r = O(1)$ and we have bounded the sum by a geometric series. For the high probability result, observe that by definition of the y_j's

$$\mathbf{Pr}(\exists j : |S_j| > y_j) = \mathbf{Pr}(\exists 1 \le j \le r : |S_j| > y_j).$$

Lemma 1.2 implies that $z_j \leq y_j$, and therefore we may apply Lemma 1.4 to conclude that this probability is $o(n^{-c})$.

APPENDIX III
CALCULATING VARIOUS QUANTITIES OF INTEREST

This Appendix describes the calculator that we use in Section IX. It is fairly easy to implement, although some care is required to ensure that the computation is efficient and that the memory requirement is reasonable.

A. Performing the MHT Calculations

Consider an MHT with tables $T_1, \ldots, T_d$, where T_i has size m_i. Let S_0 be a set of size n items hashed into the MHT, and for $i = 1, \ldots, d$, let S_i be the set of items not placed in $T_1, \ldots, T_i$. We show how to compute the individual marginal distributions of the $|S_i|$'s. In particular, this allows us to compute the crisis probability of the MHT: $1 - \mathbf{Pr}(|S_d| = 0)$.

First, note that conditioned on $|S_{i-1}|$, the distribution of $|S_{i-1}| - |S_i|$ is the same as the distribution of the number of nonempty bins resulting from randomly throwing $|S_{i-1}|$ balls into m_i bins. Letting $p_{j,m,b}$ denote the probability that randomly throwing j balls into m bins yields exactly b nonempty bins, we have that for $b = 1, \ldots, m$,

$$p_{j,m,b} = p_{j-1,m,b-1}(1 - (b-1)/m) + p_{j-1,m,b}(b/m). \quad (3)$$

Letting $P_{j,i}[b] = p_{j,m_i,b}$ for $b = 0, \ldots, m_i$, we can now compute the individual marginal distributions of the $|S_i|$'s using the following pseudo-code:

Set $\mathbf{Pr}(|S_i| = \ell) = 0$ for $i = 0, \ldots, d$ and $\ell = 1, \ldots, n$.
Set $\mathbf{Pr}(|S_0| = n) = 1$.
for $i = 1$ to d **do**
  Set $P_{0,i}[0] = 1$ and $P_{0,i}[j] = 0$ for $j > 0$.
  for $j = 1$ to n **do**
    Compute $P_{j,i}$ from $P_{j-1,i}$ using (3).
    for $\ell = 1$ to j **do**
      $\mathbf{Pr}(|S_i| = \ell) \mathrel{+}= \mathbf{Pr}(|S_{i-1}| = j) \cdot P_{j,i}[j - \ell]$

Of course, $\mathbf{Pr}(|S_{i-1}| = j)$ is typically negligible for sufficiently large j, so we can optimize the pseudo-code to greatly reduce the number of iterations of the second loop.

B. The Failure Probability of the Single Filter Summary

Consider an instance of the single filter summary of Section VIII-A with m cells and k hash functions. We show to compute upper bounds on the various failure probabilities of the summary that we believe to nearly tight when those probabilities are small.

We continue to use the notation of Appendix III.A. First, note that if $|S_{i-1}| = j$ and $|S_i| = \ell$, then the probability that a particular item of type i yields a failure is

$$(1 - (1 - k/m)^{\ell})^k \triangleq q_\ell$$

and so the conditional probability that any type i failure occurs is at most $(j - \ell)q_\ell$, by a union bound. We believe that this bound is very tight when q_ℓ is small, since other Bloom filter results [12] suggest that these $j - \ell$ potential failures are almost independent, and the union bound is extremely accurate for independent events with very small probabilities.

Now, the conditional distribution of $|S_{i-1}| - |S_i|$ given $|S_{i-1}|$ is the same as the distribution of the number of nonempty bins resulting from randomly throwing $|S_{i-1}|$ balls into m_i bins. Therefore, given that $|S_{i-1}| = j$, the probability that any type i failure occurs is at most

$$\sum_{\ell=1}^{j-1} P_{j,i}[j - \ell](j - \ell)q_\ell \triangleq f_{i,j} \quad (4)$$

and so the overall probability that any type i failure occurs is at most $\sum_{j=1}^{n} \mathbf{Pr}(|S_{i-1}| = j) f_{i,j} \triangleq f_i$. By another union bound, the total failure probability is at most $\sum_{i=1}^{d-1} f_i \triangleq f$. Once again, we believe that this bound is fairly tight when the f_i's are small.

We can now compute bounds on the various failure probabilities with the following pseudo-code:

for $i = 1$ to $d - 1$ **do**
  Set $f_i = 0$.
  for $j = 1$ to n **do**
    Compute $P_{j,i}$ from $P_{j-1,i}$ using (3).
    Compute $f_{i,j}$ using (4).
    $f_i \mathrel{+}= \mathbf{Pr}(|S_{i-1}| = j) \cdot f_{i,j}$
  Compute $f = \sum_{j=1}^{d-1} f_i$.

As in Appendix III-A, $\mathbf{Pr}(|S_{i-1}| = j)$ is typically negligible for sufficiently large j, so we can optimize the pseudo-code to greatly reduce the number of iterations of the second loop.

C. The Failure Probability of the Multiple Bloom Filter Summary

Consider an instance of the multiple Bloom filter summary of Section VIII-B with filters $B_0, \ldots, B_{d-1}$, where B_i has b_i bits and k_i hash functions. We show to compute estimates of the various failure probabilities of the summary that we believe to nearly tight upper bounds when those probabilities are small and the number n of items is large.

We continue to use the notation of Appendix III-A. First, note that if $|S_{i-1}| = j$ and $|S_i| = \ell$, then the probability that a particular item of type i yields a failure is[4]

$$(1 - (1 - k_i/b_i)^{\ell})^{k_i} \triangleq q_{\ell,i},$$

and so the conditional probability that any type i failure occurs is at most $(j - \ell)q_{\ell,i}$, by a union bound. As in Appendix III-B, we believe that this bound is tight when $q_{\ell,i}$ is small.

[4]Technically, this formula is only valid if we use *partitioned* Bloom filters, as in the single filter summary. Unfortunately, designing a summary is usually easier if we use unpartitioned Bloom filters, as in our examples in Section IX. These two varieties of Bloom filters are asymptotically equivalent, however, and a partitioned Bloom filter usually has a higher false positive probability than its unpartitioned counterpart [5]. Therefore, we expect this formula to give a nearly tight upper bound.

Now, the conditional distribution of $|S_{i-1}|-|S_i|$ given $|S_{i-1}|$ is the same as the distribution of the number of nonempty bins resulting from randomly throwing $|S_{i-1}|$ balls into m_i bins. Therefore, given that $|S_{i-1}| = j$, the probability that any type i failure occurs is at most

$$\sum_{\ell=1}^{j-1} P_{j,i}[j-\ell](j-\ell)q_{\ell,i} \triangleq f_{i,j} \tag{5}$$

and so the overall probability that any type i failure occurs is at most $\sum_{j=1}^{n} \mathbf{Pr}(|S_{i-1}| = j) f_{i,j} \triangleq f_i$. By another union bound, the total failure probability is at most $\sum_{i=1}^{d-1} f_i \triangleq f$. Once again, we believe that this bound is fairly tight when the f_i's are small.

We can now compute (approximate) bounds on the various failure probabilities using essentially the same pseudo-code as in Appendix III-B; we simply use (5) instead of (4).

REFERENCES

[1] Y. Azar, A. Broder, A. Karlin, and E. Upfal, "Balanced allocations," *SIAM J. Comput.*, vol. 29, no. 1, pp. 180–200, 1999.

[2] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[3] A. Broder and A. Karlin, "Multilevel adaptive hashing," in *Proc. 1st ACM-SIAM Symp. Discrete Algorithms (SODA)*, 1990, pp. 43–53.

[4] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP Lookups," in *Proc. IEEE INFOCOM*, 2001, pp. 1454–1463.

[5] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[6] J. Byers, J. Considine, and M. Mitzenmacher, "Geometric generalization of the power of two choices," in *Proc. 16th ACM Symp. Parallel Algorithms and Architectures (SPAA)*, 2004, pp. 54–63.

[7] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: an efficient data structure for static support lookup tables," in *Proc. 15th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, 2004, pp. 30–39.

[8] E. D. Demaine, T. Jones, and M. Pătraşcu, "Interpolation search for non-independent data," in *Proc. 15th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, 2004, pp. 522–523.

[9] P. C. Dillinger and P. Manolios, "Fast and accurate bitstate verification for SPIN," in *Proc. 11th Int. SPIN Workshop on Model Checking of Software (SPIN)*, 2004, pp. 57–75.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area Web cache sharing protocol," *IEEE/ACM Trans. Networking*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[11] G. H. Gonnet and R. B. Yates, *Handbook of Algorithms and Data Structures In Pascal and C*, 2nd ed. Reading, MA: Addison-Wesley, 1991.

[12] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better Bloom filter," in *Proc. 14th Annu. Eur. Symp. Algorithms (ESA)*, 2006, pp. 456–467.

[13] A. Kirsch and M. Mitzenmacher, "Simple summaries for hashing with multiple choices," in *Proc. 43rd Annu. Allerton Conf. Communication, Control, and Computing*, 2005.

[14] M. Mitzenmacher, "The power of two choices in randomized load balancing," Ph.D. dissertation, Univ. California, Berkeley, 1996.

[15] M. Mitzenmacher, "How useful is old information?," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 1, pp. 6–20, 2000.

[16] M. Mitzenmacher, "Compressed Bloom Filters," *IEEE/ACM Trans. Networking*, vol. 10, no. 5, pp. 613–620, May 2002.

[17] M. Mitzenmacher, A. Richa, and R. Sitaraman, , P. Pardalos, S. Rajasekaran, J. Reif, and J. Rolim, Eds., *The Power of Two Choices: A Survey of Techniques and Results*. Norwell, MA: Kluwer, 2001, pp. 255–312.

[18] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 2005.

[19] A. Pagh, R. Pagh, and S. Srinivas Rao, "An Optimal Bloom Filter Replacement," in *Proc. 16th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, 2005, pp. 823–829.

[20] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended Bloom filter: An aid to network processing," in *Proc. ACM SIGCOMM*, 2005, pp. 181–192.

Adam Kirsch (S'07) received the Sc.B. degree in mathematics–computer science (*magna cum laude*) from Brown University, Providence, RI, in 2003, and the S.M. degree in computer science from Harvard University, Cambridge, MA, in 2005. He is currently working towards the Ph.D. degree in computer science at Harvard University.

His current research interests are primarily in the applications of techniques from theoretical computer science, including the probabilistic analysis of computer processes and the design and analysis of algorithms and data structures for massive data, data streams, and long-term systems.

Mr. Kirsch received an NSF Graduate Research Fellowship in 2004.

Michael Mitzenmacher (M'99) received the Ph.D. degree in computer science from the University of California, Berkeley, in 1996.

He is a Professor of computer science in the Division of Engineering and Applied Sciences at Harvard University. His research interests include Internet algorithms, hashing, load balancing, power laws, erasure codes, error-correcting codes, compression, and bin-packing. He is the author, with Eli Upfal, of the textbook *Probability and Computing: Randomized Algorithms and Probabilistic Analysis* (Cambridge University Press, 2005).

Dr. Mitzenmacher was the co-recipient of the 2002 Information Theory Society Paper Award for his work on low-density parity-check codes.