Simple Summaries for Hashing with Multiple Choices

Adam Kirsch* Michael Mitzenmacher†

Division of Engineering and Applied Sciences
Harvard University
33 Oxford St.
Cambridge, MA 02138
{kirsch,michaelm}@eecs.harvard.edu

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.

1 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 [9, 15].

Recently, in the context of routers, Song et al. [21] 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. [21] 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?" 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. [21] consists of a counting Bloom filter [11, 16]. We review this construction in detail in Section 3.

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

*Supported in part by an NSF Graduate Research Fellowship, NSF grant CCR-0121154, and a grant from Cisco.
†Supported in part by NSF grant CCR-0121154 and a grant from Cisco.

that they are very simple to analyze; while [21] 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, 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.

2 Related Work

There is a great deal of work on multiple-choice hashing schemes [17] and on Bloom filters [2, 5]. See those references for more background.

As mentioned in the introduction, our starting point in this paper is the work of Song et al. [21], 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 3.

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 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 a multilevel hash table, and so we give a more thorough description of multilevel hash tables in Section 5.

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* [8], 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* for items not in the set. That is, a query for an item not in the set may 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 from 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 [8, 19] suggest that we must take advantage of the characteristics of our specific problem setting (for example, the skew of the distribution of the values) in order to guarantee good performance.

3 The Scheme of Song et al. [21]

For comparison purposes, we review the summary scheme of [21] 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, fully random) hash functions to d of the 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. Song et al. appear to strive for parameters that guarantee a maximum load of 1 with high probability, although their approach could be used more generally.

This basic scheme is not particularly effective. To improve the scheme, Song et al. give heuristics in order 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 [21] for details.

4 Separating Hash Tables and Their Summaries

Following Song et al. [21], 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. The cost of this separation is additional hashing. With the approach of [21], 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, so this price seems like a reasonable one to pay for 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 [21]. 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. As before, this additional cost will be outweighed by what we will gain.

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.

5 The Broder-Karlin 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 $c_1 \geq 1$ 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 $c_1 n/(1-c_2)$).

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 $T_i[h_i(x)]$. 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 modifying the original analysis, we can show that for reasonable values of c_1 and c_2 (for example, $c_1 = 10$ and $c_2 = 1/2$), the probability that there is any crisis is polynomially small. (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). We omit the formal statement and proof for reasons of space.

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 generally does very well in the face of deletions [17].

5.1 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 have 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

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

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

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

Note that the inequalities are quite tight for reasonable values of 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 $z << 1$, 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 next sub-table, using the distribution from the previous sub-table. This requires 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 [7]. Some care is required to ensure that the computation is efficient and that the memory requirement is reasonable, but the procedure is not difficult to implement.

5.2 Skew in Multilevel Hash Tables

We have mentioned that MHTs have a strong *skew property*, in the sense that the first sub-table contains a lot of the items, the second sub-table contains a lot of the rest, and so on. This can be seen by experimenting with various values in (1). However, the property is more cleanly presented using the presentation in [3] (even though this approach might give a looser approximation).

We proceed as follows. 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 $1/m_i$. Linearity of expectation now tells us that

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

Using the heuristic approximation that each $|S_i|$ is at most the bound on its expectation in the above formula, it is not difficult to show that the $|S_i|$'s decay doubly exponentially for certain reasonable choices of c_1 and c_2. Indeed, this is the property used in [3] to show that MHTs perform well with $d = O(\log\log n)$. In our case, however, this property also tells us to expect the distribution of the $|S_i|$'s to be very skewed.

6 One Summary Approach: Interpolation Search

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 [12], which requires only $O(\log\log n)$ operations on average. Insertions and deletions are trivial; simply add or remove the appropriate string.

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. The number b must therefore be chosen to be large enough to make this event extremely unlikely. And, of course, b must also be chosen to be large enough so that the probability of a false positive is also quite small. (We note that two items with the same bit string do not actually yield a failure 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.)

These probabilities 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 still have everything fit into a 64 bit word; 3 bits is enough for 8 hash functions, which should be suitable for most implementations. Setting $n = 100000$ 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 $n = 10000$, 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 implied 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. Nevertheless, there are some great advantages of this summary construction over the others discussed in this paper, most notably the ability to handle insertions and deletions easily and the very small false positive probability.

7 Bloom Filter-based MHT Summaries

In this section, we propose summaries that exploit the *skew* property of MHTs. In particular, we make extensive use of the theory of Bloom filters. For now we consider insertions only, deferring our discussion of deletions until Section 7.2. 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 very high probability, they correctly identify the sub-tables storing each of the n items.

Our first Bloom filter-based MHT summary can be seen as a simple Bloomier filter meant to allow 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. With this classification of errors in mind, the analysis of this structure follows easily from the standard analysis of a Bloom filter [5].

Lemma 7.1. *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.[1]

In general, Lemma 7.1 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

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

In our setting, Lemma 7.1 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

$$k = (\ln 2)\cdot m/\beta_1 n.$$

(This is not likely to be an integer; one of the neighboring integer values will be optimal.) 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

$$k = (\ln 2)\cdot m/n,$$

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 hash table and the corresponding summary structure. Specifically, one can vary the number of sub-tables and their sizes in the hash table, as well as the size of the summary and the number of hash functions used. Generally, the more hash functions used in the hash table, the smaller the probability of a crisis (up to some point), but increasing the number of hash functions 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, including packing multiple values together. For example, three cells taking values in the range $[0,5]$ can be packed into a 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, however, are very appealing, and can be made quite small by taking advantage of skew.

7.1 On Skew, and An Improved Construction

Lemma 7.1 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.

Under the construction above, the most likely failure to occur is a type 1 failure; there are many fewer items of types greater than 1, and correspondingly there is very little probability for a failure

[1] Technically, we may wish to differentiate between the false positive probability and the false positive rate, as defined in [13], but the distinction is unimportant in practice. See [13] for an explanation.

for these items. A natural way to reduce the probability of a type 1 failure is to introduce more skew, specifically 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 in our constructions used for experiments below.

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 $B_0, B_1, \ldots, B_{t-1}$).

Because of the skew, each successive filter can be smaller than the previous one without compromising the total failure probability. The skew in our setting 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 7; we provide an example in Section 8. 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 or hashes of items (using interpolation search as in Section 6).

The only downside of this approach is that it can require significantly more hashing than the others. For many applications this may not be a bottleneck. Also, the number of hashes required for a Bloom filter can be dramatically reduced using techniques related to double hashing, so that only two hash functions are required per (sufficiently large) filter [10, 13]. Because we are aiming for very small false positive probabilities, it is not immediately clear whether these techniques can be applied in this context; this remains an area for future work.

7.2 Deletions

Handling deletions is substantially more difficult than handling insertions. For example, the scheme proposed in [21] 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 suggest two alternative ways to handle deletions: one general, and one specific to our construction.

A natural, general approach is to use *lazy* deletions. That is, we keep a 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 bit corresponding to b to 1. When looking up an item, we treat it as deleted if we find it in a bucket whose corresponding bit is 1. When a preset number of deletions occurs, or after a preset amount of time, we can reconstruct the entire data structure (that is, the hash table, the bit array, and the summary) from scratch using the items in the hash table, leaving out the deleted items. If we allow at most αn deleted items and n undeleted items at any time, we simply must build our data structures to be able to cope with $(1+\alpha)n$ items. 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.

For MHTs, unlike other multiple-choice hashing schemes, we can delete an item simply by removing it from the hash table; no repositioning of elements is necessary. The summary, however, must be modified to reflect the deletion. In order handle such changes to the single filter summary, we modify it so that each cell contains one counter for each type, and a counter tracks the number of items in the hash table of the corresponding type that hash to the cell containing the counter. When a deletion occurs, we simply decrement the appropriate counters. A similar modification works equally well for the construction with multiple filters, replacing each Bloom filter by a counting Bloom filter.

Keeping counters over multiple types can be expensive, in terms of storage. Therefore, the interpolation search summary of Section 6 becomes more appealing when deletions must be handled. However, with very high probability, most counter values in the filter-based summaries will be extremely small, so by appropriately packing bits, counters can be competitive with the interpolation approach.

8 Numerical Evaluation

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. [21]. We use 'k' to represent 1000 and 'm' to represent one million. The summaries used in our constructions handle insertions only; we leave considerations of the extra memory required for deletions as subsequent work. This is a fair comparison against the scheme in [21], which requires additional structures to handle deletions.

For the MHT summaries, our preliminary goal was to use at most 6 buckets per item; this was less than $1/2$ the size of the hash table (in terms of buckets) than in [21], and seemed a reasonable goal. This led to the following underlying MHTs. For 10k items, there are 5 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 5.1). 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 7, 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, using standard probabilistic techniques and the same sort of calculations as described in Section 5.1, it is possible to efficiently compute upper bounds on these probabilities, and we have built a calculator for this purpose. We believe that our upper bounds are fairly tight when the probabilities are very small, and so we use them as if they were the actual probabilities, even though they may be overestimates.

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 7 hash functions for the first filter and 49 for each of the others.[2] For 100k items, we configure the first Bloom filter based summary to have 1.2m cells 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 1. For the last column, 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 the expense of a fairly large summary. The single filter scheme appears comparable to the structure of [21] for 10k items, but uses much less space; the multiple filter scheme allows further space gains in the summary, 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-

[2]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.

(a) 10k items

Scheme	Buckets	Summary Space	False Positive Probability	Failure + Crisis Probability
Song et al. [21]	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	Buckets	Summary Space	False Positive Probability	Failure + Crisis Probability
Song et al. [21]	?	?	?	?
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}

Table 1: The hash table size (in buckets), storage requirement (in bytes), false positive probability, and failure plus crisis probability for each of the schemes. IS denotes the interpolation search scheme of Section 6, SF denotes the single filter scheme of Section 7, and MBF denotes the multiple Bloom filter scheme of Section 7.1. We configure the interpolation search summary according to the examples in Section 6. The notation '?' for the Song et al. [21] summary denotes information not available in that work. All storage requirements for our summaries include the space for the bit table mentioned in Section 4.

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 have left a '?' in our table of results. We note that they report having found no crisis in one million trials.

9 Experimental Validation

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 simulation tests, we simulated random hashing by fixing hash values for each item using a standard 48-bit pseudorandom number generator.

We have simulated the single filter for 10k items in Table 1; 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.

10 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 [21]. We believe the fact that our summaries can be analyzed to bound performance is a useful characteristic that will ease adoption.

There are several possible additions to this work that we plan to pursue. Our results can be extended easily to the case where buckets can hold multiple items, and we plan to complete both analyses and simulations for this case. More experimentation should be done to test our summary structures for applications, including large-scale tests with hash functions commonly used in practice. A detailed analysis of deletion workloads to determine the effect of and best approach for deletions would be worthwhile.

References

[1] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal of Computing* 29:1, pp. 180-200, 1999.

[2] B. Bloom. Space/time tradeoffs in in hash coding with allowable errors. *Communications of the ACM*, 13(7):422-426, 1970.

[3] A. Broder and A. Karlin. Multilevel adaptive hashing. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 43-53, 1990.

[4] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP Lookups. In *Proceedings of IEEE INFOCOM*, pp. 1454-1463, 2001.

[5] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485-509, 2004.

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

[7] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang. On the false-positive rate of Bloom filters. Submitted. `http://cg.scs.carleton.ca/~morin/publications/ds/bloom-submitted.pdf`

[8] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp 30-39, 2004.

[9] M. Dahlin. Interpreting stale load information. *IEEE Transactions on Parallel and Distributed Systems*, 11(10), pp. 1033-1047, 2000.

[10] P. C. Dillinger and P. Manolios. Fast and accurate bitstate verification for SPIN. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software (SPIN)*, pp. 57-75, 2004.

[11] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281-293, 2000.

[12] G.H. Gonnet and R. Baeza-Yates. Handbook of Algorithms and Data Structures In Pascal and C, Second Edition. Addison-Wesley Publishing Company, 1991.

[13] A. Kirsch and M. Mitzenmacher. Building a better Bloom filter. Harvard University Computer Science Technical Report TR-02-05, 2005. `ftp://ftp.deas.harvard.edu/techreports/tr-02-05.pdf`.

[14] M. Mitzenmacher. The power of two choices in randomized load balancing. Ph. D. thesis, U.C. Berkeley, 1996.

[15] M. Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1), pp. 6-20, 2000.

[16] M. Mitzenmacher. Compressed Bloom Filters. *IEEE/ACM Transactions on Networking*, 10(5):613-620, 2002.

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

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

[19] A. Pagh, R. Pagh, and S. Srinivas Rao. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 823-829, 2005.

[20] M. V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237-1239, 1989.

[21] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended Bloom filter: an aid to network processing. In *Proceedings of ACM SIGCOMM*, pp. 181-192, 2005.