Improving Retrieval Performance for Invertible Bloom Counters

David Ding
fding@college.harvard.edu

Matt Rauen
rauen@college.harvard.edu

***Abstract*—We present a probabilistic data structure that represents a multi-set, supporting insertion, deletions, queries, and listing operations. We demonstrate how extra metadata can improve the recoverable capacity while holding space constant.**

I. Introduction

Bloom filters are well-known data structures that support probabilistic set membership queries and dynamic insertion of items with constant-time insertion and query times and $O(n)$ space, where n is the number of elements in the set. We consider a generalization of Bloom filters to multisets. This data structure, which we call an invertible Bloom counter, supports multiple insertions of a key and deletions (up to the number of times the key was stored). A constant-time query operation can return the multiplicity of a key in the set with some probability. These functionalities are exactly the functionalities provided by spectral Bloom filters [3].

The most important aspect of our Bloom counter, however, is the ability to list all the keys that are contained in the multiset along with their multiplicities, similar to the listing function of invertible Bloom lookup tables [1]. As with Bloom lookup table, the listing function can successfully recover all the keys with high probability if the number of keys stored, n, is less than cm, where m is the number of buckets and c some constant depending on the number h of hash functions used, and the listing fails to recover most keys with high probability if $n > cm$. We present three Bloom counters and find, theoretically and empirically, the critical threshold c.

This problem is useful in several contexts. For instance, we could imagine a router or firewall that maintains a count of the number of requests sent by different IP addresses so that problematic IP addresses (which might issue malicious messages or attempt a denial of service attack) could be identified. In this context, it is important to have fast insertion times and low space requirements, and the slow listing operation could be performed offline by another computer.

The paper is organized as follows. In section II, we discuss related works, including theoretical analysis of the number of k-cores in hypergraphs and the analysis of invertible Bloom lookup tables. In section III, section IV, and section V, we present theoretical analysis for standard Bloom counters (based off invertible Bloom lookup tables), Bloom counters that store additional hash sums, and Bloom counters that store additional keys. Each of these sections describe the data structure, the listing operation, and analyzes the threshold of recoverability. Finally, in section VI, we present empirical results on the performance of these Bloom counters.

The main contribution of this paper is the description of additional metadata that could be added to each bucket of the Bloom counter to improve the recoverability threshold holding space constant. We show that by storing one extra key in each bucket of the hash table, we can increase the number of recoverable keys per byte.

II. Related Work

Spectral Bloom filters are generalizations of Bloom filters that represent dynamic multisets [3]. For a multiset with n distinct keys and N keys with multiplicity, a spectral Bloom filter using $k \log N + O(n)$ bits will return the correct multiplicity of a key with failure probability exponentially small in k. The spectral Bloom filter stores a counter at each bucket. To insert a key, the key is hashed to h different locations, and the counter at each bucket is incremented. To retrieve the multiplicity of a key, we take minimum of the counters at each of the h locations the key is hashed to. This procedure will never underestimate the multiplicity of a key, and will return the right answer if one of the h buckets only contains that key.

The idea of retrieving the stored keys is discussed in Goodrich and Mitzenmacher [1], which presents a generalization of Bloom tables to store key-value pairs. The invertible Bloom lookup tables, supports a listing operation, which succeeds with very high probability if the number of keys is below $c(h)m$, where m is the number of buckets and $c(h)$ is a number (usually close to 1) that depends on h. The simplest version of invertible Bloom lookup tables (which assumes all keys are inserted only once and no extraneous deletion occurs) works by storing a count at each bucket representing the number of keys stored at the bucket, along with the sum of all keys stored at the bucket and a sum of all values. If any bucket has count one, we know that there is a unique key value pair stored at that bucket and hence the keysum and the valuesum must equal that key value pair. We retrieve that entry and delete the key from the table, which might decrease the count of other buckets to equal one. We continue until all buckets have count 0 or until all nonempty buckets have count at least 2. In the former case, we successfully recovered all the key value pairs in the table; in the latter case, we fail to recover some pairs. This approach could be augmented to guard against multiple insertions and extraneous deletions by storing sums of the hashes of the keys and the values (using a different hash function than that used to hash the key into the table) and performing consistency checks on these hash sums.

The theoretical analysis of the success probability of listing hinges on the observation that the listing operation is precisely

the same algorithm used to see if a hypergraph contains a k-core. An h-uniform hypergraph is a collection of m vertices and h-edges, which are size h subsets of these vertices; a regular graph is a 2-uniform hypergraph. A k-core is a sub-hypergraph in which every vertex has degree at least k. The invertible Bloom lookup table can be regarded as a h-hypergraph (where h is the number of hash functions) with m vertices representing the buckets. Each edge, consisting of h vertices, corresponds to the h locations that any key can hash too. Then, a degree 1 vertex represents a bucket with count 1, and we remove edges that contain a vertex of degree 1, until we arrive at a graph with no vertices or until we reach a 2-core. Hence, listing is successful if and only if the corresponding hypergraph has no 2-core (assuming that no two keys hash to the exact same h buckets).

Very tight thresholds have been found for the existance of k-cores [2]. In particular, it has been proven that

Theorem II.1. *If $k, h \geq 2$ with k, h not both equal to 2, then a random hypergraph chosen uniformly among hypergraphs with $\frac{c}{h!}$ edges will have a k-core with probability $p \to 1$ (as $m \to \infty$) if $c > c*$ and probability $p \to 0$ (as $m \to \infty$) if $c < c^*$, where*

$$c^* = \min_{x>0} \frac{x(h-1)!}{1 - e^{-x} \sum_{i=0}^{k-2} \frac{x^i}{i!}}.$$

III. Standard Invertible Bloom Counters

We first describe a straightforward combination of the ideas behind spectral Bloom filters and invertible Bloom lookup tables.

A. Description the Data Structure

In this data structure, we store in each bucket the following items.

1) *count* - the total number of keys (not necessarily distinct)
2) *keysum* - the sum of all keys hashed
3) *hashsum* - representing the sum of all the hashes of the keys hashed to this bucket (using a different hash function than the ones used to hash keys into buckets).

Note that without the two sum fields, this data structure is identical to spectral Bloom filters.

Insertion, deletion, and querying is straightforward. To insert a key, we hash the key to h buckets and increment the count in each bucket, add the key to the keysum of each bucket, and add the hash of the key to the hashsum of each bucket. To delete the key, we decrement the count of each bucket the key hashes to, subtract the key from the keysum, and subtract the hash of the key from the hashsum. To query for the multiplicity of a key, we return the minimum of the count fields for all the buckets the key hashes to. Because these three operations are identical to those of spectral Bloom filters, the same bounds on the correctness probability of querying holds for our data structure. For simplicity, we assume that we never delete an item more times than it is stored; however, the analysis presented in Goodrich and Mitzenmacher on handling extraneous deletions will apply for our data structure also, since we store a hashsum at each bucket.

h	Threshold
3	$0.818m$
4	$0.773m$
5	$0.701m$
6	$0.637m$
7	$0.582m$

TABLE I: Threshold for the existence of 2-cores, i.e., for the success of listing

B. The Listing Operation

Algorithm 1 Algorithm for listing

```
Set repeat to True
Set recoveredlist to []
while repeat do
    Set repeat to False
    for all Buckets b do
        if b.keysum/b.count is an integer then
            Set k = b.keysum/b.count
            if all buckets k hashes to have count at least b.count then
                Add (key, b.count) to recoveredlist
                Delete b.count copies of key from the table
                Set repeat = True
return recoveredlist
```

The main idea of the listing operation is the same as for invertible Bloom lookup tables: we wish to identify a bucket that stores only one distinct key. Then, the keysum for that bucket will just be the key, and the count will be the multiplicity of the key. The problem is that we allow for multiple insertions of a key, so that our data structure is not equivalent to an invertible Bloom lookup table where the value is just the multiplicity of the key. Because keys can be inserted multiple times, it is impossible to know whether a newly inserted key is distinct from every other key stored in the bucket unless we store every key hashed to the bucket. Hence, it is impossible to tell if a bucket contains exactly one distinct key unless we store every key already.

Nevertheless, we can identify the buckets that store one distinct key with high probability using the hashsum. We observe that if a bucket stores one distinct key, then keysum/count must be the value of that key, and the hashsum must equal $\text{count} \times \text{hash}(\text{keysum}/\text{count})$. Moreover, the probability that $\text{count} \times \text{hash}(\text{keysum}/\text{count})$ if the bucket contains more than 1 key is very low. Hence, we can use this condition as a good indicator for whether or not a bucket contains exactly one distinct key. The pseudocode for the algorithm is given in Algorithm 1.

Supposing that our criteria never fails, the analysis for these invertible Bloom counters is identical to the analysis of invertible Bloom lookup tables given in Goodrich and Mitzenmacher [1]. In particular, listing fails only if there is a 2-core in the corresponding hypergraph, and using Theorem II.1, we see that a 2-core exists with high probability if the number of distinct items is greater than cm for some constant c depending on h, the number of hash functions. Similarly, no 2-cores exist with high probability if the number of distinct items is less than cm. This threshold is given in I

C. Bounds on the Failure of a Particular Attempt

Unlike invertible Bloom failures, listing could also fail if the criterion for determining that a bucket contains a unique key fails. In this section, we bound the probability of this occurring.

Suppose the bucket contains keys x_1, x_2, ..., x_k with frequency f_1, ..., f_k. Moreover, suppose that our hash function H was chosen uniformly at random from a universal family of hash functions. Then, we want the probability of

$$H\left(\frac{f_1x_1+\cdots+f_kx_k}{f_1+\cdots+f_k}\right)=\frac{f_1H(x_1)+\cdots+f_kH(x_k)}{f_1+\cdots+f_k}.$$

Suppose that $\bar{x}=\frac{f_1x_1+\cdots+f_kx_k}{f_1+\cdots+f_k}$ does not equal any of the x_i. Then, $H(\bar{x})$ is independent of $H(x_i)$, so this probability is just $\frac{1}{S}$, where S is the size of the image of the hash function. Now, suppose $\bar{x}$ equals one of the x_i, say x_1 without loss of generality. Then,

$$H(\bar{x})=H(x_1)=\frac{f_1H(x_1)+\cdots+f_kH(x_k)}{f_1+\cdots f_k}$$

which can be simplified to give

$$H(\bar{x})=H(x_1)=\frac{f_2H(x_2)+\cdots+f_kH(x_k)}{f_2+\cdots f_k}.$$

Now, the x_i are distinct, so $H(\bar{x})$ is independent of the $H(x_i)$ for $i\neq 1$, and the probability is still $\frac{1}{S}$.

In the listing operation, the condition can only be triggered once per operation. Triggering the condition once for one bucket will give another bucket another chance for the condition to be triggered, so the total number of opportunities for triggering the condition is upper bounded by $m+(m-1)+\cdots+1=\frac{m(m+1)}{2}$. Hence, by the union bound, the probability of reporting an incorrect key for the entire listing process is bounded by $\frac{m(m+1)}{2S}$. Since $S=2^b$ if b is the number of bits required to store a hash value, using $k\log_2 m$ bits for the hash value, we can achieve error probability of less than m^{2-k}. In practice, we need some more space for the hashsum to prevent overflow problems.

IV. Even-Odd Keysums

With our first modification, we partition the keys into two classes, and store a keysum for each class in each bucket. The intuitive benefit is that we no longer block on a 2-core if there exists a bucket in the 2-core in which the bucket has only two keys, and those two keys are in distinct classes. However, to preserve total space, we have to scale the number of buckets down by a factor of $3/4$.

A. Modifications to the Data Structure

In our implementation, we need an additional hash function that maps keys to $\{0,1\}$, independently and uniformly at random. The modified implementation of the bucket contains the following elements:

1) *hashsum* - A hashsum.
2) *keysum0* - A keysum of the keys that hash to 0.
3) *keysum1* - A keysum of the keys that hash to 1.
4) *count* - A count.

When we insert a key k, we iterate over the h buckets that it hashes into, add $\mathit{hash}(k)$ to the hashsum, add one to the count, and add the key to the appropriate keysum (depending on its hash to $\{0,1\}$). Deleting a key is simply the complementary operation.

B. The Listing Operation

We start by looking for non-empty buckets in which either *keysum0* or *keysum1* is 0. Upon finding such a bucket, we guess that the bucket only contains a single key, and that key is k =(*keysum0* + *keysum1*)/*count*. To check this hypothesis, we then see if

$$\mathit{hash}(k)*\mathit{count}=\mathit{hashsum}.$$

If this is true, we delete *count* copies of k from the data structure, and continue.

If we have no buckets for which one of the keysums is zero, yet there are still non-empty buckets, we iterate over the buckets, theorizing that some bucket might contain exactly two keys, which hash to distinct values in $\{0,1\}$. Thus, for a particular bucket, we check if there exists some $i\in\{1,2,\ldots,\mathit{count}-1\}$ such that we may have i copies of the key that hashes to 0, and $\mathit{count}-i$ copies of the key that hashes to 1. That is, we guess that we have the following keys:

$$k=\mathit{keysum0}/i$$
$$k'=\mathit{keysum1}/(\mathit{count}-i).$$

To see if this is possible, we check whether

$$\mathit{hash}(k)*i+\mathit{hash}(k')*(\mathit{count}-i)=\mathit{hashsum}.$$

Whenever we succeed, we delete i copies of k and $(\mathit{count}-i)$ copies of k', and then return to looking for non-empty buckets in which either *keysum0* or *keysum1* is 0. If we iterate over all of the buckets and are unable to perform such a deletion, then we fail.

C. Bounds on the Recovery Threshold

In this section, we come up with an approximation for the number of keys that can be inserted into the data structure before we fail with significant probability. Note that if we fail at recovery, then we must deadlock in a scenario where each bucket contains at least two distinct keys that both hash to 0 or 1. With at most one key of both types, we could make progress, as the above algorithm explains.

In the following analysis, note that we use mini-bucket to refer to either the subset of keys in a bucket that hash to 0, or the subset of keys that hash to 1. To begin the analysis, we start by computing the probability that a particular mini-bucket gets at least two keys is given by

$$1-\left(1-\frac{h}{2m}\right)^n-n\left(\frac{h}{2m}\right)\left(1-\frac{h}{2m}\right)^{h(n-1)/2m}$$
$$\geq 1-e^{-hn/2m}-\frac{nh}{2m}e^{-h(n-1)/2m}.$$

Thus, in expectation, we can insert

$$\frac{2m}{h}\cdot\left(1-e^{-hn/2m}-\frac{nh}{2m}e^{-h(n-1)/2m}\right)$$

keys into the data structure such that every mini-bucket with at least two keys gets a third key hashed to the same location. In this scenario, we have at least three keys in a particular bucket if and only if one of the mini-buckets had at least two of the original keys.

To see this, note that if we had at least two keys in the same bucket, then this approach introduces a third. On the other hand, if we did not have two keys in the same mini-bucket, no key gets added, so we have at most two keys between the two mini-buckets. Thus, our old failure condition is exactly the same as there being a 3-core in the new graph.

To continue the analysis, we consider the threshold for seeing a 3-core among the n inserted keys. This is given by

$$\frac{c^*}{h!} \cdot m = \min_{x>0} \frac{xm}{h\left(1 - e^{-x}(1+x)\right)^{h-1}}.$$

For particular values of h, we get the following thresholds:

h	Threshold
2	$1.675m$
3	$1.553m$
4	$1.334m$
5	$1.158m$
6	$1.022m$
7	$0.915m$

Thus, to find the actual thresholds, we can solve the equation given by

$$\frac{2m}{h} \cdot \left(1 - e^{-hn/2m} - \frac{nh}{2m} e^{-h(n-1)/2m}\right) + n = T_h,$$

where T_h is the threshold for a 3-core for a given h. This gives the following results:

h	Threshold
2	$1.301m$
3	$1.196m$
4	$1.029m$
5	$0.896m$
6	$0.793m$
7	$0.712m$

Note that these are upper bounds on the thresholds, since there is a significant chance that the number of keys will lead to more collisions than in expectation, which represents a significant chance of failure. Indeed, when comparing these values against the empirical results, we see that they overstate the actual bound by approximately $0.1m$, and demonstrate that as h grows, this method becomes more and more ineffective.

D. Bounds on the Failure of a Particular Attempt

Given that it is possible to recover the multiset from the bloom filter, we would like to bound the probability that any particular list operation fails.

This failure can only occur if we attempt to remove a key which is either not present in the bloom filter, or is present with a smaller multiplicity than what we are trying to remove. Each of these conclusions is associated with doing a comparison against the hashsum, so if S denotes the size of the image of *hash*, we fail with probability at most $1/S$ on any particular check. Therefore, if we can bound the total number of checks that we do in the entire algorithm, via union-bound, we can bound the total probability of failure.

Consider the number of checks that we run on any particular bucket. We will bound the number of checks on a bucket before a key is removed from it. Let K be the number of distinct keys in the bucket, and *count* be the actual count stored in the bucket. As long as we don't remove a key, we do one check to see if there is exactly one key, and $(\textit{count} - 1)$ checks to identify the two keys. Hence, we do at most *count* checks before a key is deleted. The worst case for the total number of checks is when each key occurs once in the bucket, so we do

$$\sum_{i=1}^{\textit{count}} i = \frac{\textit{count}(\textit{count}+1)}{2}$$

total checks. The total number of checks across all buckets is at its worst when all of the keys hash into the same buckets, since this function is convex, which implies that as long as

$$S \geq \textit{count}^2,$$

we fail with probability at most $1/2$. We can make this arbitrarily small by repeatedly rerunning the algorithm with a different permutation of the order in which we consider the buckets. In practice, this is also significantly better.

V. Keeping a Key

Here we describe an alternative in which we store an extra key in each bucket to aid with recovery. In order to maintain a constant amount of space, we reduce the number of buckets to $3/4$ of the original capacity.

A. Modifications to the Data Structure

We modify the data structure so that each bucket contains the following elements:

1) *count* - the total number of keys in this bucket (not necessarily distinct)
2) *keysum* - the sum of all keys in this bucket.
3) *hashsum* - the sum of all the hashes of the keys in this bucket.
4) *onekey* - Some key that was inserted into the bucket.

When we insert a key k, we iterate over the h buckets that it hashes to, and add $\textit{hash}(k)$ to the hashsum, k to the keysum, and 1 to the count, as in the standard implementation. Moreover, we store one key in this bucket. There are several possibilities on which key to choose:

1) The first key ever inserted to this bucket;
2) The last key inserted to this bucket;
3) A random key inserted into this bucket. If we replace the currently stored key with probability $1/\textit{count}$, every inserted key (including duplicates) is chosen with equal probability. This means that keys that were inserted more often will more likely be stored.

Note that onekey is not necessarily contained in a bucket if we delete the key; onekey is only necessarily inserted into the bucket once.

B. The Listing Operation

For any given bucket, we have three methods of guessing a key contained in a bucket.

1) We guess that onekey is (still) contained in the bucket. We check that $keysum = onekey * count$ and $hash(onekey) * count = hashsum$, and if so, we return onekey and count.
2) We check if $keysum/count$ is an integer, and if so, check if $hash(keysum/count) * count = hashsum$. If this condition is met, we return keysum/count and count.
3) We guess that onekey is one of the keys contained in the bucket and iterate over all partitions $i + j = count$, where i ranges from 1 to count-1. We check if $(count - i * onekey)/j$ is an integer, and if so, we set $k = (count - i * onekey)/j$. We then check if $i * hash(onekey) + hash(k) * j = hashsum$, and if so, we return onekey with count i.

These conditions are ordered in terms of decreasing certainty in correctness; the first condition is correct with higher probability than the second, and the second condition has higher probability of being correct than the third.

In order to implement the listing operation while keeping in mind the different certainties of the three conditions, we keep track of a three-level priority queue of buckets representing which buckets to recover first.

1) If a bucket is currently in the first queue, we check the first condition, and if it holds, we add $(onekey, count)$ to the list of recovered keys and delete *count* copies of *onekey* from the data structure.
2) If we are in the second queue, we check to see if the second condition holds, and if so, add (keysum/count, count) to the list of recovered keys and delete *count* copies of *keysum*/*count* from the data structure.
3) If in the third queue, we check the third condition. If it holds, we add (onekey, i) to the list of recovered keys and delete i copies of onekey from the data structure.

If we fail on a bucket in queue i, we move it to queue $i + 1$. If it is already in queue 3, we move it to the end of queue 3. Any time we delete from a bucket, we move it back up to queue 1. To pick the next bucket to be checked, we choose the bucket at the head of the non-empty queue with smallest index, which ensures that we are inferring with the lowest probability of failure at any given step. If we are unable to make any progress after cycling through the entirety of queue 3, we then consider the possibility that one of the stored keys would allows us to make progress (as in the third recovery approach) if it were stored in one of the other buckets that it hashes to. In pseudocode, this is:

Note that upon termination, we resume the recovery algorithm from the earliest non-empty queue, as above. We can improve the probability of successful recovery via the following two optimizations:

1) Whenever we delete a key from the data structure via the listing operation, set the key to be -1 everywhere that key is stored as *onekey*.

Algorithm 2 Using onekeys globally.

```
for all remaining non-empty buckets do
    Set k to be the key stored in the bucket
    if k != -1 then
        Set B to be the buckets that k hashes to
        for all buckets b in B do
            for all i in {1,2,...,b.count-1} do
                Set i' to be b.count - i
                Set k' to be (b.keysum - k*i)/i'
                Set hashsum to be hash(k)*i+hash(k')*i'
                if hashsum == b.hashsum then
                    for all buckets b' in B do
                        Delete i copies of k from b'
                    terminate
```

2) Before listing, build up a list L for each bucket of the onekeys that hash there. Then, we make the following modification for condition 3: we iterate through the keys k in L, iterate through partitions $i + j = count$, and check if there is an integer k' satisfying $i * k + j * k' = keysum$ and $i * hash(k) + j * hash(k') = hashsum$, and if so, we return k, i.

We found that empirically, these optimizations significantly improves the listing success rate.

We can logically extend condition 3 by checking to see if there's a bucket in which there are two *onekeys* and one other key, the if there are three *onekeys* and one other key, etc. This modification takes significantly longer (there are $\Theta(count^d)$ partitions of count into d integers), but it increases the threshold as described in the following section.

C. Bounds on the Recovery Threshold

Consider the scenario in which we are guaranteed to be unable to recover the graph, independent of the choices that we make in recovery. If we have any bucket in which we have at most one key that does not appear as a *onekey* across all buckets, then we will be guaranteed to make progress via the last stage of the algorithm (if we use the extension of the listing algorithm discussed at the end of section **??**). Thus, we only stop being able to proceed if all remaining non-empty buckets contain at least two keys that are not *onekeys*. This is equivalent to there being a 2-core on the keys that are not stored as *onekey*s. We now compute this threshold, and evaluate it on small values of h.

We wish to find the number of distinct onekeys stored in the Bloom counter. We assume that the onekey is set for the first key inserted into the bucket. Let X_i be the indicator random variable which equals 1 if the i-th distinct key inserted into the structure is stored as a onekey. We then wish to compute the expected value $\mathbb{E}[X_1 + \cdots + X_n]$.

Now, X_i equal 1 if and only if one of the h buckets the i-th key hashes to is empty. Let p_j equal the probability that at least j of the h buckets are empty. Then, by the principle of inclusion exclusion, $\Pr(X_1 = 1) = \sum_{j=1}^{h} (-1)^{j-1} \binom{h}{j} p_j$.

To compute p_j, we note that the probability that j buckets are empty is the probability in the balls and bins model that

h	Threshold
3	$1.429m$
4	$1.294m$
5	$1.158m$
6	$1.045m$
7	$0.952m$

TABLE II: Theoretical thresholds for the one-key Bloom counter

$(i-1)h$ balls (representing the previously inserted keys) avoid j of the m bins, so this probability is

$$p_j = \left(1 - \frac{j}{m}\right)^{ih} \leq e^{-ihj/m}.$$

Plugging this all in, we get the expected number of one keys that are stored to be

$$\begin{aligned}
\mathbb{E}[X_1 + \cdots + X_n] &= \mathbb{E}[X_1] + \cdots + \mathbb{E}[X_n] \\
&= \sum_{i=1}^{n}\sum_{j=1}^{h}(-1)^{j-1}\binom{h}{j}e^{-ihj/m} \\
&= \sum_{j=1}^{h}(-1)^{j-1}\binom{h}{j}\sum_{i=1}^{n}e^{-ihj/m} \\
&\leq \sum_{j=1}^{h}(-1)^{j-1}\binom{h}{j}\sum_{i=1}^{\infty}e^{-ihj/m} \\
&= \sum_{j=1}^{h}(-1)^{j-1}\binom{h}{j}\frac{1}{1-e^{-hj/m}} \\
&= \sum_{j=1}^{h}(-1)^{j-1}\binom{h}{j}\frac{1}{1-(1-hj/m+\cdots)} \\
&\approx \sum_{j=1}^{h}(-1)^{j-1}\binom{h}{j}\frac{m}{jh} \\
&= \left(\sum_{j=1}^{h}(-1)^{j-1}\binom{h}{j}\frac{1}{jh}\right)m
\end{aligned}$$

Let C be the coefficient of m in the above expression. We want to identify the threshold such that inserting $n - Cm$ keys into the data structure does not generate a 2-core, and inserting n keys does not generate a 3-core with high probability. Recall that by Theorem II.1, if $n - Cm \leq \frac{c^*}{h!}m$, we expect no 2-cores. Hence, the threshold for recoverability for one-key Bloom counter is given by $\frac{c^*}{h!} + C$. The numerical values for these values is given in Table II. Note that these thresholds are for the Bloom counters that iterates through all partitions of keysum into onekeys stored in that bucket; in practice, looping through these partitions takes time $count^d$, where d is the number of onekeys associated with the bucket, and this is too slow. The simplified algorithm discussed in detail in section **??**.

By the above analysis, since $1.429 \geq (4/3) \cdot 0.818$, it follows that this approach has a higher threshold for the same space complexity as the standard solution when $h = 3$. Similarly, we can see that it offers improvements for $4 \leq h \leq 7$.

D. Bounds on the Failure of a Particular Attempt

As in the analysis of even-odd, we bound the number of incorrect checks that can be done, and then by union bound, have a bound on the overall probability of failure. Note that this analysis is for the unmodified algorithm, so we stop before we consider subsets of *onekey*s of size at least two. As we see experimentally, this does not lower the threshold significantly, but allows for tighter bounds on the failure of any particular list attempt.

Consider the number of checks that we run on any particular bucket. We will bound the number of checks on a bucket before a key is removed from it. Let K be the number of distinct keys in the bucket, and *count* be the actual count stored in the bucket. As long as we don't remove a key, we do one check in the first queue, one check in the second queue, and at most $K \cdot (\mathit{count} - 1)$ checks in the third queue, since the worst case is where all K keys are explicitly stored somewhere, and we do $\mathit{count} - 1$ checks on each key. This makes a total of

$$K \cdot (\mathit{count} - 1) + 2$$

checks before we remove a key. In the worst case, each key has multiplicity 1, so *count* decreases by only 1 between each removal. This implies a total of at most

$$\sum_{i=1}^{\mathit{count}} i(i-1) + 2 = (\mathit{count}^3 + 5\mathit{count})/3$$

checks on this particular bucket. Thus, if we choose our hash such that $S \geq N^3$, where N is the total number of insertions, we fail with probability at most $1/3$ (as this is maximized when the count is concentrated in a single bucket). We can make this arbitrarily small by repeatedly rerunning the algorithm with a different permutation of the buckets in the first queue.

VI. Empirical Results

We implemented the three different versions of the bloom counter using Python. Our code generated around 10000 distinct keys chosen uniformly randomly between 1 and 10000000 and inserted each key into the bloom counter n times, where n is chosen uniformly randomly between 1 and 20. After the insertion, we list the items stored in the bloom counter, and we say that the bloom counter recovered key k correctly if it listed k with its exact frequency. We ran this experiment, varying the number of buckets m used in the bloom counter, and kept track of the value of m that triggers failed recovery. The results of this simulation are listed in Table III. Figure 1 shows a more detailed look at the successful probability for $h = 4$ hash functions. We obtained hash functions from [4] and [5].

We see that for the standard Bloom counters, the empirical threshold is exactly the same as predicted by theory, which agrees with the results in Goodrich and Mitzenmacher [1]. Moreover, we see that in the one-key Bloom counter, the empirical threshold worse than theoretical threshold, which is expected since the theoretical thresholds assumes that we iterate through all partitions of keysum into the onekeys (stored in the entire table) that hash to that bucket, whereas in our implementation, we only partition keysum into one onekey and an unknown key. Despite this, we see that the one-key Bloom

h	threshold
3	0.81
4	0.76
5	0.70
6	0.63
7	0.58

(a) The empirical threshold for standard Bloom counters.

h	threshold
3	1.22
4	1.08
5	0.90
6	0.65
7	0.54

(b) The empirical threshold for even-odd Bloom counters.

h	threshold	improvement
3	1.27	1.13
4	1.14	1.12
5	1.01	1.08
6	0.90	1.04
7	0.81	1.05

(c) The empirical threshold for one-key Bloom counters without the optimizations discussed at the end of section V-B. The improvement column is the ratio of the critical number of keys that can be stored in the extra-key bloom counter as compared to the critical number of keys that can be stored in a standard bloom counter of the same space (i.e., with $\frac{4}{3}$ times more buckets).

h	threshold	improvement
3	1.34	1.24
4	1.21	1.18
5	1.07	1.16
6	0.95	1.13
7	0.86	1.11

(d) The empirical threshold for one-key Bloom counters with the optimizations discussed at the end of section V-B.

TABLE III: Empirical thresholds for the three described methods.

counter performs better than the standard Bloom counter using an equivalent amount of space (i.e. with $\frac{4}{3}m$ buckets, since a bucket for standard Bloom counters stores 3 items and a bucket for one-key Bloom counters stores 4). We also note that the optimizations discussed at the end of section V-B significantly increased the threshold for which we are able to perform successful listing.

The situation with even-odd Bloom counters is more complex. When the number of hash functions is small, the even-odd Bloom counters perform better than the standard Bloom counters and than theory. However, the performance declines rapidly with the number of hash functions. This is because the even-odd Bloom counters have a significantly higher rate of returning a key that was not inserted, or returning a key with multiplicity greater than the number of times the key was inserted. Such an error causes the Bloom counter to enter an undefined state, decreasing the chance of successful recovery. The problem is more significant with larger values of h since each key is inserted into more buckets, resulting in more buckets with nonzero counts that could report containing a key that does not exist.

VII. Conclusion

In this paper, we introduced an invertible Bloom counter data structure, combining ideas from invertible Bloom lookup tables and spectral Bloom filters. We considered modifications based on partitioning the keysum into two halves, as well as storing an extra key in each bucket. We found that by storing an extra key at each location, we achieve theoretical improvements in the amount of keys that can be stored with the same amount of total space, and saw these claims verified experimentally.

In particular, we saw that when we use our one-key variation, in which an extra key is stored in each bucket, we get an 11% to 24% improvement in the number of keys that we can store in the same amount of space if the number of hash functions is between 3 and 7.

Possible approaches to take in the future include combining these two modifications to see if the resulting data structure still leads to improvements. Moreover, we could investigate for larger values of h to see if there is a threshold number of hash functions after which we stop seeing improvements.

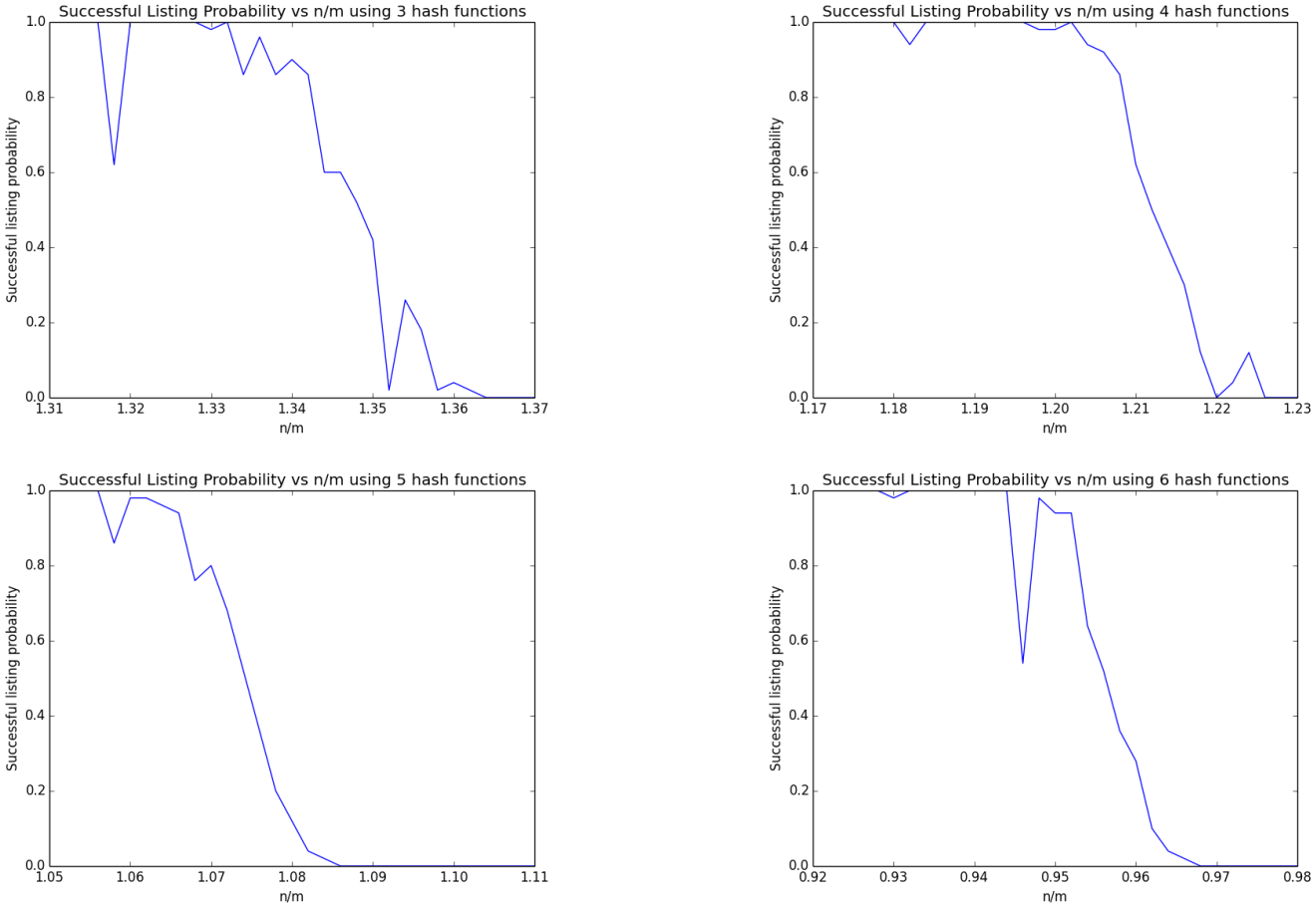

Fig. 1: Probability of successful listing vs $\frac{n}{m}$ with $h = 3, 4, 5, 6$ hash functions. We see a rapid drop of probability around 1.21, which is the threshold reported in Table III

APPENDIX A
APPENDIX

Here we provide the coded implementation for the computation of thresholds:

```
import Queue as queue
import hashlib
import random
import sys

# hash functions from
# http://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-
# that-accepts-an-integer-hash-key
# and
# http://eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx
def hash0(x):
    x = ((x >> 16) ^ x) * 0x45d9f3b
    x = ((x >> 16) ^ x) * 0x45d9f3b
    x = ((x >> 16) ^ x)
    return x

def hash1(x):
    return x

def hash2(x):
```

```
    # We mod out by a prime so that hash2 and hash1 do not always have same parity.
    return (x * 2654435761) % 104729

def hash3(x):
    # FNV hash
    h = 2166136261
    h = (h * 16777619) ^ x
    return h

def hash4(x):
    # One at a time (Bob Jenkins)
    x = (x << 10) ^ (x >> 6)
    x += (x << 3)
    x ^= (x >> 11)
    x += (x << 15)
    return x

def hash5(x):
    # Rotating hash
    x = x * 2654435761
    x = (x << 4) ^ (x >> 28)
    x = x * 104729
    x ^= (x >> 12)
    return x

def hash6(x):
    return int(hashlib.sha512('0'+str(x)).hexdigest()[:8], 32)

def hash7(x):
    return int(hashlib.sha512('1'+str(x)).hexdigest()[:8], 32)

def hash8(x):
    return int(hashlib.sha512('2'+str(x)).hexdigest()[:8], 32)

bucket_hash = hash4

hashes = [hash6, hash8, hash7, hash3, hash0, hash1, hash2]

class NaiveBucket(object):
    def __init__(self):
        self.count = 0
        self.hash_sum = 0
        self.key_sum = 0
        self.retrievalMethods = 1

    def store(self, key, count = 1):
        self.count += count
        self.hash_sum += count * bucket_hash(key)
        self.key_sum += count * key

    def delete(self, key, count = 1):
        assert self.count >= count
        self.count -= count
        self.hash_sum -= count * bucket_hash(key)
        self.key_sum -= count * key

    def space(self):
        return 3

    def retrieve(self, val):
        if val < 0 or val >= self.retrievalMethods:
            return None
```

```
        if val == 0:
            if self.key_sum % self.count == 0:
                key = self.key_sum / self.count
                if bucket_hash(key) * self.count == self.hash_sum:
                    return key, self.count
        return None

class Bucket(object):
    def __init__(self):
        self.count = 0
        self.hash_sum = 0
        self.key_sum = [0, 0]
        self.retrievalMethods = 2

    def store(self, key, count=1):
        self.count += count
        self.hash_sum += count * bucket_hash(key)
        self.key_sum[bucket_hash(key) % 2] += count * key

    def delete(self, key, count=1):
        assert self.count >= count
        self.count -= count
        self.hash_sum -= count * bucket_hash(key)
        self.key_sum[bucket_hash(key) % 2] -= count * key

    def space(self):
        return 4

    def retrieve(self, val):
        if val < 0 or val >= self.retrievalMethods:
            return None
        if val == 0:
            if self.key_sum[0] == 0 or self.key_sum[1] == 0:
                keysum = sum(self.key_sum)
                if keysum % self.count == 0:
                    key = keysum/self.count
                    if (bucket_hash(key) * self.count == self.hash_sum
                      and self.key_sum[bucket_hash(key) % 2] == keysum):
                        return key, self.count
        if val == 1:
            for i in range(1, self.count):
                j = self.count - i
                if self.key_sum[0] % i == 0 and self.key_sum[1] % j == 0:
                    key1 = self.key_sum[0] / i
                    key2 = self.key_sum[1] / j
                    if bucket_hash(key1) * i + bucket_hash(key2)*j == self.hash_sum:
                        return key1, i
        return None

class Bucket2(object):
    def __init__(self):
        self.count = 0
        self.hash_sum = 0
        self.key_sum = 0
        self.onekey = -1
        self.retrievalMethods = 3

    def store(self, key, count=1):
        self.count += count
        self.hash_sum += count * bucket_hash(key)
        self.key_sum += count * key
        if self.onekey == -1:
```

```
            self.onekey = key
        else:
            if random.random() < 0.5:
                self.onekey = key

    def delete(self, key, count=1):
        assert self.count >= count
        self.count -= count
        self.hash_sum -= count * bucket_hash(key)
        self.key_sum -= count * key
        if self.key_sum == 0:
            self.onekey = -1

    def space(self):
        return 4

    def retrieve(self, val):
        if val < 0 or val >= self.retrievalMethods:
            return None

        if self.onekey == -1: return None

        if val == 0:
            if (self.onekey * self.count == self.key_sum
              and bucket_hash(self.onekey) * self.count == self.hash_sum):
                return self.onekey, self.count

        if val == 1:
            if self.key_sum % self.count == 0:
                key = self.key_sum / self.count
                if bucket_hash(key) * self.count == self.hash_sum:
                    return key, self.count

        if val == 2:
            for i in range(0, self.count):
                j = self.count - i
                if (self.key_sum - j * self.onekey) % i == 0:
                    key1 = (self.key_sum - j * self.onekey)/i
                    key2 = self.onekey
                    if bucket_hash(key1) * i + bucket_hash(key2)*j == self.hash_sum:
                        return key1, i

        return None

class Counter(object):
    def __init__(self, capacity=1000, bucketclass = NaiveBucket, num_hashes = 3):
        self.table = [bucketclass() for i in range(capacity)]
        self.capacity = capacity
        self.retrievalMethods = self.table[0].retrievalMethods
        self.num_hashes = num_hashes

    def get_bucket(self, i, v):
        divisions = self.capacity / len(hashes)
        return self.table[divisions * i + (hashes[i](v) % divisions)]

    def get_buckets(self, v):
        # indices = set(hashes[i](v) % self.capacity for i in range(len(hashes)))
        # return [self.table[i] for i in indices]
        return [self.get_bucket(i, v) for i in range(self.num_hashes)]

    def store(self, value, count=1):
        for b in self.get_buckets(value):
```

```
        b.store(value, count)

def delete(self, value, count=1):
    for b in self.get_buckets(value):
        b.delete(value, count)

def space(self):
    return len(self.table) * self.table[0].space()

def count(self):
    return sum(b.count for b in self.table)

def retrieve(self):
    '''Retrieve is destructive'''
    retlist = []

    queues = [queue.Queue() for i in range(self.retrievalMethods)]
    empty = set()

    for b in self.table:
        queues[0].put(b)

    while True:
        try_again = True
        last_queue = True
        for i in range(self.retrievalMethods-1):
            last_queue = False
            if not queues[i].empty():
                b = queues[i].get()
                if b.count == 0:
                    empty.update([b])
                    break
                a = b.retrieve(i)
                if a is not None:
                    key, count = a
                    assert count >= 0
                    if all(
                      self.get_bucket(j, key).count >= count
                      for j in range(self.num_hashes)):
                        for j in range(self.num_hashes):
                            self.get_bucket(j, key).delete(key, count)
                            queues[0].put(self.get_bucket(j,key))
                        retlist.append((key, count))
                        break
                queues[i+1].put(b)
                break
            if i == self.retrievalMethods - 2:
                last_queue = True
        if last_queue:
            try_again = False
            if queues[-1].empty():
                return retlist
            length = queues[-1].qsize()
            for i in range(length):
                b = queues[-1].get()
                if b.count == 0:
                    empty.update([b])
                    continue
                a = b.retrieve(self.retrievalMethods-1)
                if a is not None:
                    key, count = a
                    assert count >= 0
```

```
                        if all(self.get_bucket(j, key).count >=
                               count for j in range(self.num_hashes)):
                            for j in range(self.num_hashes):
                                self.get_bucket(j, key).delete(key, count)
                                queues[0].put(self.get_bucket(j,key))
                            retlist.append((key, count))
                            try_again = True
                            break
                    queues[-1].put(b)
            if not try_again:
                break

        return retlist

def simulate(num_hashes, capacity, klass):
    true_count = {}

    different_keys = 10000

    for i in range(different_keys):
        true_count[random.randint(1, 10000000)] = random.randint(1, 20)

    counter =
      Counter(capacity=int(len(true_count)/capacity),
              bucketclass=klass,
              num_hashes=num_hashes)

    # print 'Number of distinct keys:', len(true_count)
    # print 'Number of hash functions:', len(hashes)

    for k in true_count:
        counter.store(k, true_count[k])
    ret = set(counter.retrieve())
    ideal = set(true_count.items())
    return len(ideal-ret)

def frange(a, b, dx):
    c = a
    while c > b:
        yield c
        c += dx

lookup = {
    'Standard': {
        3: 0.85,
        4: 0.8,
        5: 0.75,
        6: 0.66,
        7: 0.62,
    },
    'Even-Odd': {
        3: 1.3,
        4: 1.25,
        5: 1.2,
        6: 1.15,
        7: 0.55,
    },
    'Extra key': {
        3: 1.3,
        4: 1.25,
        5: 1.2,
        6: 1.15,
```

```
        7: 1.1,
    }
}

numtrials = 50

def count(l):
    return sum(1 for i in l if i == 0)

for name, klass in [('Even-Odd', Bucket), ('Extra key', Bucket2)]:
    print name
    print r'h & threshold \\'
    for r in range(3, 8):
        for threshold in frange(lookup[name][r], 0.4, -0.01):
            trials = [simulate(r, threshold, klass) for i in range(5)]
            if sum(1 for i in range(5) if trials[i]==0) >= 2:
                print r'%d & %.2f\\' % (r, threshold)
                break
```

REFERENCES

[1] M. Goodrich and M. Mitzenmacher. Invertible Bloom Lookup Tables. *CoRR*, abs/1101.2245, 2011.

[2] M. Molloy. The pure literal rule threshold and cores in random hypergraphs. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 672-681. Society for Industrial and Applied Mathematics, 2004.

[3] S. Cohen and Y. Matias. Spectral Bloom Filters. In SIGMOD 2003, p 241-252.

[4] http://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key

[5] http://eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx