Counting with TinyTable: Every Bit Counts!

Gil Einziger Roy Friedman
Computer Science Department, Technion
Haifa 32000, Israel
{gilga,roy}@cs.technion.ac.il

***Abstract*—Counting Bloom filters (CBF) and their variants are data structures that support membership or multiplicity queries with a low probabilistic error. Yet, they incur a significant memory space overhead when compared to lower bounds as well as to (plain) Bloom filters, which can only represent set membership without removals.**

This work presents TinyTable, an efficient hash table based construction that supports membership queries, multiplicity queries (statistics) and removals. TinyTable is more space efficient than existing alternatives, both those derived from Bloom filters and other hash table based schemes. In fact, when the required false positive rate is smaller than 1%, it is even more space efficient than (plain) Bloom filters.

I. Introduction

Bloom filters and their many variants can provide low memory approximation for problems such as set membership [20], approximate counting [32], approximate state machines [5], and even arbitrary function approximation [11].

These constructions are an important enabling factor for many database applications such as approximate aggregate queries, iceberg queries, bifocal sampling, range queries and distributed join [12]. The latter reduces the utilized bandwidth when performing a distributed join query. Bloom filter variants also have multiple networking applications such as accounting, monitoring, load balancing, routing, security policy enforcement and caching [5], [7], [11], [14], [15], [16], [18], [20], [21], [37]. Many of these networking applications are representative of general stream processing problems.

These capabilities are also very useful in many well known systems, e.g, Google's BigTable [10], Google Chrome [1], Apache's Hadoop [29], Facebook's (Apache) Casandra [28], Squid (web proxy cache) [2] and Venti (archive system) [35].

Yet, variants of Bloom filters that support removals or can represent multisets and statistics accounting consume much more space than plain Bloom filters that can only represent (simple) set membership without removals. Some more space efficient hash table based alternatives have been also proposed [6], [7], [27]. However, these cannot support multiset (and statistics) representation. Moreover, their memory requirements still leave much room for improvement.

A. Contribution

In this paper, we present *TinyTable*, a novel hash table based construction that can represent set membership with removals as well as multiset membership (and statistics) and at the same time is significantly more space efficient than previously proposed alternatives.

We present three different configurations that vary in space efficiency and operation speed. We show that all these configurations outperform Bloom filters for query operations and that two of them also offer faster update operations.

Our work also includes a new definition of error for multiset queries. We analyze this error and show through simulations that our analysis is accurate and that our error definition gives meaningful insight.

B. Paper Organization

Section II introduces the main concepts behind approximate representations and surveys existing approaches for implementing them. We describe TinyTable in Section III followed by a formal analysis in Section IV. Next, we present performance measurements that both explore TinyTable's characteristics and compare it with leading alternatives in Section V. Finally, we conclude with a discussion in Section VI.

II. Preliminaries and Related Work

A. Bloom Filters and Variants

A Bloom filter [4] is a very simple and space efficient data structure that supports approximate set membership queries. However, Bloom filters do not support removals or multiplicity queries and have a non local memory access pattern. Extending the functionality of Bloom filters is a common approach in the literature.

Counting Bloom filters (CBF) [20] are a natural extension of Bloom filters in order to support removals. CBF are almost identical in operation to Bloom filters, except that instead of bits they employ counters. The add operation uses k different hash functions, but instead of setting k different bits in a Bloom filter, it increments the k counters in the counting Bloom filter, effectively counting the number of collisions for each element of the array. The remove operation in CBF simply decrements each of the corresponding counters, and the contain method returns true only if all the hash functions point to counters that are greater than zero.

While conceptually simple to understand and implement, counting Bloom filters that use 4 bit counters require 4 times as much space as Bloom filters. This has motivated many works on optimizing the space requirements of CBF. Specifically, [30], [31] are counting Bloom filter implementations that replace the fixed sized CBF counters with Huffman coded variable length counters. Similarly, [21] uses a hierarchial structure where overflowed counters are hashed to a smaller CBF. Alternatively, the *variable increment counting Bloom*

filter (VI-CBF) [36] increments the counters with a different value for each hash function. The values are picked from B_h sets. Small sums of B_h sets are typically unique and therefore the different hash functions rarely collide with one another.

Spectral Bloom filters [12] are similar in design to counting Bloom filters. However, they are optimized to represent multisets rather than support removals. Therefore, a spectral Bloom filter is implemented using variable counter length that is typically achieved through Elias coding. This coding is especially efficient for representation of large counters. Counting sketches techniques such as *count min sketch* [13] and *multi stage filters* [19] have similar structure, but for simplicity avoid the use of variable counter length. Unfortunately, even with the above optimizations, all Bloom filter based constructions consume a lot more space than Bloom filters. The only exception known to us is *Counter braids* [32], which is a very space efficient data structure for representation of multisets. However, counter braids requires a very long decode time and therefore does not enable read operations. This motivates the search for alternative methods such as fingerprint hash tables.

B. Approximate Set: Hash Tables

The most simple hash table based Bloom filter can be achieved with a *perfect hash function*. Such a function hashes all the items in the set without collisions, and therefore a simple array can be used to store their fingerprints [8]. In order to check whether $x \in S$, $P(x)$ is calculated and the fingerprint of x is compared to the fingerprint stored at $P(x)$. Since any element x s.t. $x \notin S$ hashes to a certain location in the array, the false positive probability in this case is the same as the probability that fingerprints of two different items are identical. That is, if the fingerprint size is $\lceil log\left(\frac{1}{\varepsilon}\right)\rceil$ bits, the false positive probability is ε. Perfect hashing can therefore be used in order to create a space optimal Bloom filter. Unfortunately, perfect hashing is not practical for dynamic workloads where the set of items is not known in advance.

The idea behind *d-left hashing* [7] is to simulate a perfect hash function using balanced allocation. That is, the table is partitioned into multiple equally sized sub-tables, also called *buckets*, where new elements are placed in the least-loaded bucket. This approach allows d-left hash tables to be dimensioned statically so that overflows are unlikely and the average load per bucket is close to the maximum load.

Rank indexed hashing [27] is an alternative approach. This technique does not use balanced allocation, and instead lets the bucket load fluctuate. Each overflowing bucket is assigned an extension bucket, and statistical multiplexing is used in order to make sure that there are enough allocated extension buckets. Exhaustive search is then used in order to find the most optimal interplay between bucket size and the required number of bucket extensions.

While hash table based approaches can provide counting Bloom filter functionality in a relatively space efficient manner, they do not provide counting functionality. To the best of our knowledge, TinyTable is currently the only fingerprint hash table that provides such functionality.

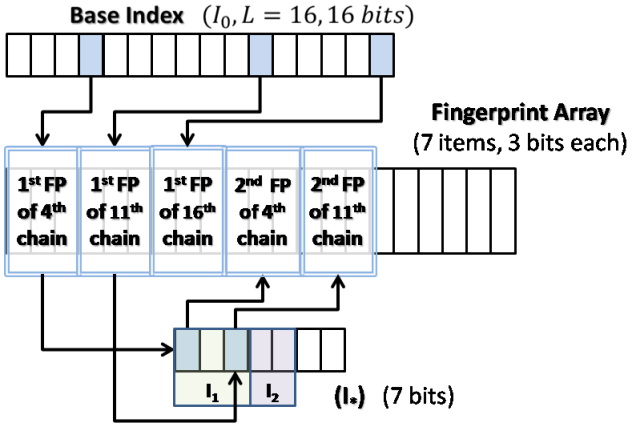


Fig. 1. An example of rank indexing technique [27]. Index state and offset are used to encode a chain based hash table. In this example, there are 3 non empty chains (4th, 11th and 16th). We first store the first item of every non empty chain. Additionally, 4th and 11th chains have second items that are stored right after the first in their chain items.

TinySet [17] is recent compact hash table based construction that is inspired by rank indexed hashing. Yet, TinySet focuses on providing a Bloom filter functionality. That is, TinySet is optimized towards an efficient representation of approximate set either without removals or with a very limited number of removals. For the specific case where no removals occur, TinySet offers better accuracy, flexibility and performance than TinyTable. In contrast, TinyTable targets the more complex approximate set with (unlimited) removals problem and approximate multiset problem. In particular, TinyTable fully supports removals and efficient statistics representation while TinySet has no such capabilities.

C. The Rank Indexing Technique

TinyTable borrows the bucket organization technique of rank index hashing, as it is efficient and well documented. The full description of this technique can be found in [27]. We provide this brief introduction for completeness and clarity of presentation.

This scheme uses a single hash function: $H : X \rightarrow B \times L \times R$. The image of the function has three parts: the first is the bucket index (B), the second (L) is the specific chain inside the bucket, and the third is a compressed fingerprint (R). In principle, to find an item in the table, the appropriate bucket and chain are accessed, and the item's fingerprint is compared against all the fingerprints in that chain.

Two indexes are used to provide a sparse chain representation; the *Base Index* (I_0) that contains 1 bit per chain, and the *Higher Index* (I_*) that contains an additional bit per fingerprint. Figure 1 shows a bucket with a few chains and the matching rank indexing bucket. The items that are first in their chain are stored first and they are ordered by chain number. These items are followed by items that are second in their chain, also ordered by chain number and so on.

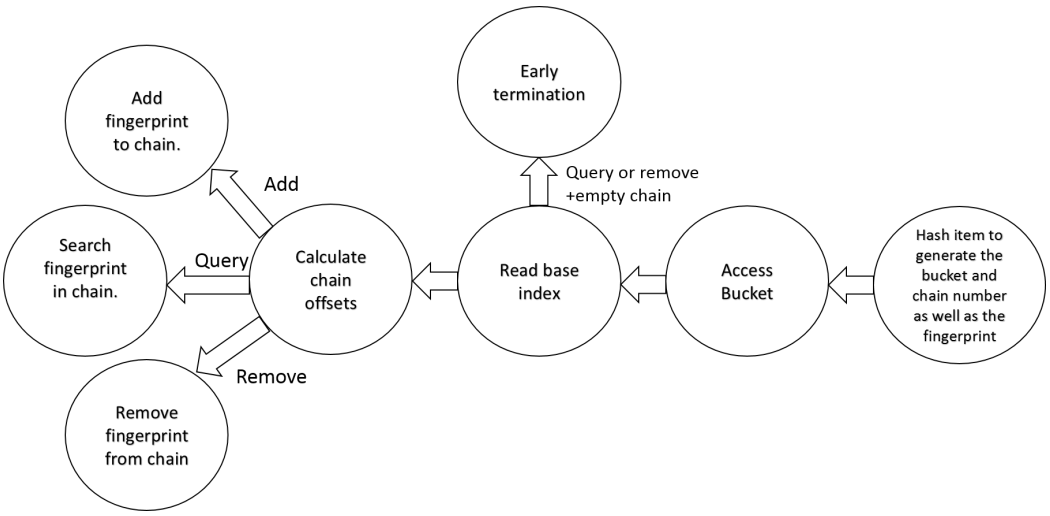


Fig. 2. A flowchart of TinyTable operations.

Every unset bit in I_0 represents an empty chain, and every set bit represents a non empty chain. A chain is accessed by calculating the memory offset of every item in the chain. These offsets are calculated using rank operations that count the number of set bits before a specific index, i.e., $rank(I_0, i)$ counts the number of set bits before the i^{th} bit. Such operations can be implemented efficiently in software with a `popcount` machine instruction that counts the number of 1s in a memory word combined with a bitwise `and` instruction.

In the example of Figure 1, there are three set bits in I_0. The size of I_1 is therefore 3 bits. The first bit of I_1 is associated with the first set bit of I_0 that is related to the 4^{th} chain. Since this bit is set, that chain has an additional fingerprint stored in it. That fingerprint is the first one among the second in their chain. $rank(I_1, 0) = 0$ discovers that the second item of the 4^{th} chain is at offset $3+0 = 3$. $rank(I_1, 3) = 2$ is calculated, to determine that I_2 contain two bits. Since the first bit of I_2 is unset, the 4^{th} chain has no additional items stored in it.

This technique also supports addition and deletion of items. However, these operations require shifting items so that their offset always matches the index.

III. TinyTable

TinyTable differs from previous work, as it employs a novel overflow mechanism. In order to handle bucket overflows, TinyTable dynamically modifies the bucket size according to the actual load in that specific bucket. However, instead of allocating new memory, the resize happens at the expense of a neighboring bucket, possibly causing that bucket to overflow as well. In that case, we repeat the process.

Unlike previous work, TinyTable can potentially use all the fingerprints array without ever overflowing. This feature exposes a new space efficiency to speed tradeoff as well as enables TinyTable to be extended to multisets with minimal impact on space efficiency.

The complexity of additions and removals increases with the table load. Specifically, TinyTable handles overflows using a *Bucket Expand* operation, which takes its inspiration from a linear probing hash table. That is, if we insert an item (T) to bucket i, and due to the insertion the bucket overflows and some other item T' has no place in bucket i, we store T' in bucket $i + 1$. We note that this bucket may overflow as it receives a new item and will repeat this operation until one of the buckets does not overflow. The challenge behind this potentially simple idea is to do so in an efficient way.

A high level overview of the operations in TinyTable is given in Figure 2. As can be seen, initially, TinyTable behaves the same for all operations. The item is first hashed, and the fingerprint is generated along with the appropriate bucket and chain id. Afterwards, the bucket is accessed. In TinyTable, the initial bucket offset is not fixed and therefore the bucket offset is calculated as we explain below. After the bucket is accessed, we first check the base index (I_0). Recall that in this index a set bit indicates a non empty chain. Therefore, we perform early termination for query and remove operations if the requested chain is empty.

In other cases, we continue and calculate the chain offsets. This offset indicates where the chain items are stored, as well as where an additional chain item should be stored. This is done according to the rank indexing technique.

Afterwards, each operation is performed differently. For example, a query operation searches for the item's fingerprint in the appropriate offset. On the other hand, an add operation adds an item at the last place of the chain and shifts the fingerprints in the bucket one place to the right. If the bucket overflows, TinyTable handles it with the bucket expand operation that is explained below.

Section III-A describes how TinyTable implements the bucket expand operation in an efficient manner. Section III-B describes an additional optimization for the counting Bloom filter functionality, while section III-C describes efficient counter representation for the approximate multiset and statistics problem. To the best of our knowledge, TinyTable is

the first fingerprint hash table solution that fully supports multiplicity queries.

A. Table Construction

1) Notations and Definitions: We denote

- N – the number of expected items,
- B – the number of buckets,
- L – the number of chains inside a bucket,
- λ – the average chain length, and
- S – the fingerprint size.

Hence, $\lambda = \left(\frac{N}{BL}\right)$, as there are N items that are inserted to B buckets, and in each bucket, an item can be placed in L different chains. The false positive in TinyTable depends only on the average chain length (λ) and the fingerprint size (S) as shown in Section IV-A.

Since there are B buckets and N items, on average $\frac{N}{B}$ items are inserted to each bucket. TinyTable requires each bucket to be initially configured to store $C > \frac{N}{B}$ fingerprints. Such a configuration ensures that there is always enough room in the table to accommodate all the items.

Definition We denote $\alpha = \frac{C}{\frac{N}{B}} = \frac{CB}{N}$ a performance parameter.

Intuitively, α measures the amount of free space in the table. When $\alpha = 1$, the table is completely full. When $\alpha = 2$, each bucket contains enough space for twice the expected number of items. That is, the table is 50% full. TinyTable exploits a space/speed tradeoff with respect to α. In particular, increasing α results in faster operations at the expense of space efficiency. TinyTable becomes more and more space efficient but slower as α approaches 1. As mentioned above, we require that $C > \frac{N}{B}$, meaning that TinyTable can operate with any $\alpha > 1$. The specific value of α can be tailored to the specific needs of different applications.

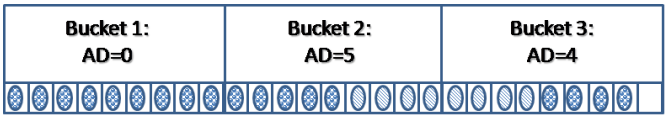
It is also important to note that query operations are indifferent of α, and are relatively fast regardless of α. The dependence of update complexity on α is analyzed in Section IV-A1.

Cell Due to the particular way in which TinyTable represents items and their associated counter values, we define a *cell* to be the amount of space required to store a single fingerprint in TinyTable. An item that appears only once consumes a single cell. Yet, an item that appears multiple times may consume multiple cells. The exact number depends on the specific encoding used, as explained below.

2) Dynamic Bucket Size With Anchor Distance Counters: We build our tables as two continuous memory chunks. The first one is used in order to store the index $(I_0, I_\star)$ and the second one is used to store the fingerprints. Each bucket is therefore defined only by the starting point of its index. Once the starting point is known, we can use rank indexed operations to read or update the table.

In TinyTable, in order for one bucket to expand, the initial location of the next bucket should be shifted. We therefore use a single counter per bucket named *Anchor Distance (AD)*.

(a) Before adding an item to Bucket 1. Anchor distance values determine the memory offset of each bucket.



(b) After adding an item to Bucket 1 it expands on account of Bucket 2 that expands on account of Bucket 3.

Fig. 3. Example of the bucket expand operation

Every time we perform an expand operation, we increment the anchor distance of the next bucket by one. That is, when we read the next bucket, we add the anchor distance to the initial bucket offset. The rank indexing technique and table operation remain the same.

An expand operation has to shift both the index of the next bucket and its fingerprints one cell to the right and update the anchor. This change enables one bucket to expand at the expense of the neighboring bucket without changing offset calculations. A high level illustration of the expand operation can be found in Figure 3. In this example, adding an item to Bucket 1 triggers two bucket expand operations: First, Bucket 1 expands on account of Bucket 2, incrementing its anchor distance in the process. Second, Bucket 2 overflows and therefore expands on account of Bucket 3 shifting it one fingerprint (i.e., one cell) to the right and incrementing its anchor distance. Since Bucket 3 does not overflow, the addition is finished.

3) Handling Anchor Distance Overflows: We now show a technique to count the number of items in a bucket, where the offset of the index is known. We start by counting all the first items in their chain by calculating $X_1 = Rank(I_0, L)$. We can then count all the second items in their chain by calculating $X_2 = Rank(I_1, X_1)$ and so on until we count all the items in the bucket. In practice, this operation is relatively fast since long chains are unlikely.

This technique can be useful for recovering overflowed anchors. As long as there is enough room in the table to accommodate all items, there is at least a single bucket that did not overflow. Therefore, if we count the items from the first previous bucket that did not overflow, we can restore the correct anchor distance of its neighboring bucket. We can continue doing so until we recover the anchor distance for the bucket we want to read or update.

In the extreme, one could suggest to eliminate anchor counters altogether in order to save space and only rely on this recovery process. However, in this case, operations' execution time becomes slow. Alternatively, we could eliminate the recovery process entirely if we use long anchor counters and achieve faster operations at the expense of slightly more space. Our approach is to carefully size these counters in order to

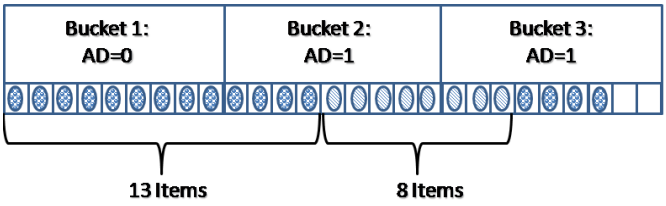


Fig. 4. Example of anchor recovery process in order to access bucket 3 when the anchor distances are only 1 bit long.

achieve the best of both worlds as explained in Section V-C.

We present an example of the anchor recovery protocol in Figure 4. In this example, anchor distance counters are only a single bit. That is, if the anchor distance reaches 1, we cannot tell if the offset is 1, or that the counter has overflowed. We therefore perform the overflow recovery technique. For example, assume that we need to access Bucket 3. Unfortunately, the anchor distance of Bucket 3 has overflowed, so we look in the previous bucket, Bucket 2. However, the anchor distance of Bucket 2 has also overflowed. Hence, we begin our offset calculation from the previous bucket, Bucket 1. Luckily the anchor distance of Bucket 1 did not overflow so we can access Bucket 1.

We count the number of stored items in Bucket 1 and discover that it has 13 items. Since each bucket is configured to have room only for 9 items, the correct anchor distance of Bucket 2 is 4. Counting the items of Bucket 2, we discover that it has 8 items. Therefore, the correct anchor distance of Bucket 3 is 3. Having calculated the correct anchor distance for bucket 3, we can now access that bucket.

Although the recovery process may be costly for crowded tables, we allocate anchors so that the vast majority of the anchor counters do not overflow. The recovery process therefore becomes shorter and less frequent.

B. Counting Bloom Filter (Without Counters)

When counting Bloom filters are used in order to represent a dynamic set, items are typically inserted to the set just once, or at most just a few times. While previous works suggested to add a small counter to each fingerprint in order to handle duplicate fingerprints, this solution is not efficient. First, this approach may result in false negatives if the fixed size counter overflows. Second, it reduces the space efficiency in order to address an event that is unlikely.

For example, consider buckets with $\lambda = 0.625$ and $L = 64$. In this configuration, there are 64 chains in a bucket, but only 40 items per bucket on average. Therefore, just $\approx 13\%$ of the chains are expected to have two or more items stored. If we also assume that the fingerprint size is 6 bits, then the probability of two identical fingerprints is $\frac{1}{2^6}$, and the overall probability that the counters are used is less than 0.5%. If we use two bit counters like previously suggested, we increase the overall space consumption in this example by $\approx 19\%$. We therefore note that when the goal of the data structure is to support a dynamic set with removals, it is more efficient to use no counters at all!

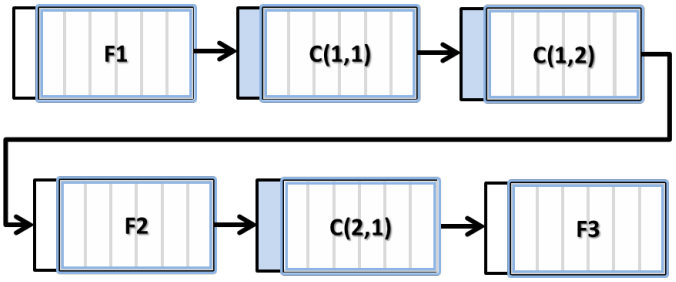


Fig. 5. A chain with six cells representing three items through fingerprints and variable sized counters. The first item (F1) has a large counter value that require two cells C(1,1) and C(1,2), while the second one (F2) only has a short counter. The third item (F3) has a counter of value of 1 and therefore has no counter cell allocated.

In this case, the table handles the case of two identical fingerprints in the same chain in the same way as it would have handled two different fingerprints in that chain. If a fingerprint is found at least once in the chain, the table will return a positive answer to a query. An interesting observation of this technique is that we now require the same space for both Bloom filter functionality and for counting Bloom filter functionality. Moreover, TinyTable is now able to provide counting Bloom filter functionality while consuming less space than a plain Bloom filter. Let us note that in this case, the number of cells consumed by each item is equal to its value.

C. Supporting Statistics – Efficient Counter Representation

When the goal of the table is to represent a multiset, it is not efficient to store the same fingerprint over and over again. In particular, if a single item appears a thousand times in the data, we obviously do not want to store a thousand fingerprints in the same chain.

In this case, a more efficient solution is to add a counter to the chain. Unfortunately, adding the same sized counter to all fingerprints is also very wasteful, since items that appear just a few times still require a very long counter.[1] We therefore use the same cells both for fingerprints and for counters. In particular, we divide the table cells into two types: *Fingerprint Cell* is a cell that is used as a fingerprint and *Counter Part* is a cell that is used as a part of a counter. In order to distinguish the two we add a single bit to each cell. This bit indicates whether this cell is a fingerprint or a counter part.

We associate counter parts and fingerprints using the order of items within the chain. To do so, we decide that each chain should always start with a fingerprint as the first cell and that counter parts are always associated to the fingerprint to the left. An example of this technique can be found in Figure 5. In this figure, we abstracted the rank indexed bucket ordering.

The first cell in the chain is a fingerprint as its first bit is white (0). This cell is followed by two counter parts, $C(1,1)$ and $C(1,2)$. Since each of them is 6 bit long, we understand

[1] Yet, an item that appears only once does not require a counter at all, since the existence of its fingerprint already serves as an indication that it appeared once.

that $F1's$ counter stores a value that requires more than 6 bits. There is another item stored in this chain, as the 4th cell in the chain is an additional fingerprint cell. That item appeared more than once, as it is also extended by a counter part. However, the second item appeared a small number of times, as it only has a single counter part. Finally, the third item only appeared once and therefore it has a fingerprint, but no counter part. If it appears again, it will be allocated a counter part. Our Java implementation [3] allocates counter cells as needed, i.e., when more counter space is needed we add a counter cell and when one of the counter cells is no longer needed we remove it.

Let us emphasize that the access to an item in TinyTable remains essentially the same here. That is, a membership query simply follows the same mechanism as detailed above for identifying the bucket and chain within the bucket. Inside the chain the query procedure scans until it either finds the corresponding fingerprint, in which case it returns true, or if it does not find the fingerprint in the chain, it returns false. Hence, since the first bit of each cell uniquely identifies a cell as either a fingerprint or counter, the code for membership queries remains exactly as without this optimization (but now the first bit of each fingerprint is always 0).

An estimation starts the same, yet when it finds the corresponding fingerprint, it continues to scan the chain until it finds the next fingerprint or end of chain. This way, it knows how many counter cells where allocated for the corresponding item (or fingerprint). If none were allocated, it returns 1. If c fingerprint cells are allocated, it returns the value represented by the combination of these c cells + 1. An add operation acts similarly, except that if the corresponding fingerprint does not exist, it adds it and terminates. If the fingerprint cell is found without any counter cells, then a counter cell is allocated and all following cells in the chain are shifted (and in case the buckets overflows, this invokes a bucket expand). If there are already counter cells allocated for this fingerprint, then they are incremented by one. If this increment causes an overflow, then an additional counter cell is allocated, which causes a shift of the following cells in the chain, etc. A remove (or decrement) operation does the opposite.

D. Overflow Handling and Statistics

It is important to understand that there is a strong connection between the fact TinyTable can correctly support statistics and its overflow handling mechanism. While this mechanism results in an unbounded worse case complexity, it remains correct even in the multiset scenario. The amount of cells used by a bucket is affected by both the average number of items that are distributed uniformly to buckets and by the number of required cells to store each item that are distributed according to the workload. Hence, the previous approach of assigning a pre-allocated bucket extension to overflowing buckets cannot exploit this optimization as it calculates the amount of required bucket extensions under the assumption of a uniform item distribution that is not true in this case.

IV. Analysis

A. False Positive in Sets

When TinyTable is used to encode sets, items are randomly inserted to buckets. We assume that items are not added multiple times to the table but false positives may still result in two identical fingerprints in the same chain. In these settings, if there are B buckets and N items, each bucket receives on average $\frac{N}{B}$ items. Since items are assigned randomly to L different chains, each chain is expected to have $\lambda = \frac{N}{BL}$ items on average.

The false positive probability, therefore, depends on the size of the stored fingerprints. In particular, if S bits are allocated per fingerprint then the probability that two different items have identical fingerprints is simply $\frac{1}{2^S}$.

Once performing a query, we compare an item's fingerprint to all the fingerprints that are stored in the corresponding chain. That is, we compare it on average to λ different fingerprints. The false positive rate is therefore $\frac{\lambda}{2^S}$. In other words, if we want to achieve a false positive rate of ε, we require that $S = \left\lceil \log\left(\frac{\lambda}{\varepsilon}\right)\right\rceil$

An important observation about this simple analysis is that the false positive rate has no dependence on α. That is, the false positive rate of TinyTable only depends on the fingerprint size (S) and the average chain length. The parameter α only affects the update operation complexity, as we show below.

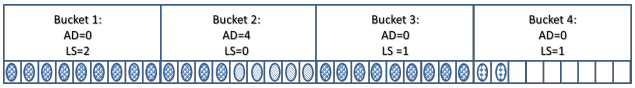
1) Operation Complexity: We now provide an intuitive bound on the update complexity of TinyTable. As mentioned above, in case of overflows, an add operation may need to resize multiple buckets. Thus, we measure the operation complexity in buckets. Moreover, query operations only read the AD counter and a single bucket index. We therefore treat their complexity as constant. Query operations only operate on multiple buckets if the AD counters overflow, but that case can be made arbitrarily rare with moderate overheads as we empirically show in the result section.

Cyclic Distance The *cyclic distance* between bucket j and bucket i is $j - i$ if $i < j$ and $j - i + B$ otherwise.

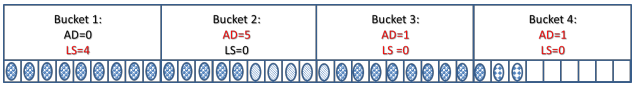
Logical Size The *logical size (LS)* of a bucket i is 0 if $AD[i] > 0$ and is the cyclic distance between j and i otherwise, where j is the next bucket s.t. $AD[j] = 0$.

Intuitively, when the AD counter of a specific bucket is 0, we say that the bucket is part of a larger "logical bucket" and therefore its logical size is 0. For buckets whose AD counter is not 0, we define their logical size as the distance to the next bucket whose AD counter is 0. For example, in Figure 6(a), Bucket 1 is of logical size 2 as its AD counter is 0. Since Bucket 3 is the next bucket with a zero AD counter, the logical size of Bucket 1 is $3 - 1 = 2$.

Figure 6(b) describes the change in structure following an addition to Bucket 1. The addition causes Bucket 1 to further expand into Bucket 2, causing it to overflow and expand into Bucket 3, which also overflows and expands into Bucket 4. In



(a) Bucket 1 overflowed and expanded into Bucket 2, which did not overflow. Its logical size is therefore 2. Bucket 2's AD counter is larger than 0 so its logical size is 0. Buckets 3 and 4 did not overflow so their logical size is 1.



(b) The update of anchor distances and logical sizes following an add to Bucket 1. Bucket 1 overflows and expands into Bucket 2, which also overflows and expands into Bucket 3, which overflows and expands into Bucket 4. Since Buckets 2-4 overflowed, their logical size is now 0. The logical size of Bucket 1 is raised from 2 to 4.

Fig. 6. An example of the logical size (LS) definition. Note that the LS is only used for analysis and is not required to operate TinyTable.

this case, the logical size of Bucket 1 is now 4, since the next bucket whose AD counter is 0 is now Bucket 1 $(1-1+4=4)$. The rest of the buckets now have larger than 0 AD counters and their logical size is therefore 0.

Bounding the number of shift operations is equivalent to bounding the logical sizes of buckets. To that end, we define a series of random variables, one per bucket, $\{X_i^{(n)}\}$. These variables are generated by the traditional balls and bins experiment where n balls are randomly thrown into B bins. $X_i^{(n)}$ in this case represents the number of balls in a bin, or in our context, the number of fingerprints that are inserted to each bucket.

It is important to understand that these random variables are not independent. For example, if we know that $X_1^{(n)} = 0$, we know that the rest of the $X_i^{(n)}$ have a larger value in expectation. Similarly, if we know the value of all other $X_i^{(n)}$s, we also know the value of $X_1^{(n)}$. Consequently, it is hard to reason about $X_i^{(n)}$s directly, forcing us to make the following detour.

Definition Define $\{Y_i^{(n)}\}$ to be a series of independent random variables generated in a Poisson experiment with success probability of $\frac{1}{B}$ and an average number of successes of $\frac{n}{B}$.

In this case, each random variable is generated by n independent coin flips that increment the variable upon success (with probability $\frac{1}{B}$). That is, it is possible that the total sum of $Y_i^{(n)}$ is not n. However, it is always n in expectation.

We would like use lemma 5.10 from the text book of Mitzenmacher and Upfal [34]:

> [**Lemma 5.10 in [34]**] Let $f(X_1,..,X_n)$ be a non-negative function such that $f\left(X_1^{(n)},...,X_B^{(n)}\right)$ is monotonically increasing in n then
>
> $$E\left(f\left(X_1^{(n)},...,X_B^{(n)}\right)\right) \leq 2E\left(f\left(Y_1^{(n)},...,Y_B^{(n)}\right)\right).$$

Our first step is to define a meaningful function that is non negative and monotonically increasing in expectancy. We suggest the following $SIZE()$ function:

$$SIZE\left(X_1^{(n)},...,X_B^{(n)}\right) = \frac{\sum_{i=1}^{B} LS\left(X_i^{(n)}\right)}{B}.$$

It is easy to verify that $SIZE()$ is non negative. It is also intuitive to see that it is monotonically increasing in expectation. Indeed, an addition may reduce the value of multiple $X_i^{(n)}$s to 0, but in that case the size of some other variable is increased by the same total amount. Therefore, the sum remains the same and the expectancy increases monotonically.

The next step is to obtain an upper bound of the expectancy of $SIZE()$. To do so, we first bound the expectancy of LS. We first start by describing the probabilistic behavior of LS with the following simple observations:

$$P\left(LS\left(X_i^{(n)}\right)=0\right) = P\left(AD\left[i\right]>0\right)$$

$$P\left(LS\left(X_i^{(n)}\right)=1\right) = P\left(AD\left[i\right]=0\right) \wedge$$
$$P\left(X_i^{(n)} > C\right) \wedge P\left(X_i^{(n)} + X_{i+1}^{(n)} \leq 2\cdot C\right)$$

$$P\left(LS\left(X_i^{(n)}\right)=k\right) = P\left(AD\left[i\right]=0\right) \wedge$$

$$\left(\bigcap_{j=i}^{j=i+k} P\left(\sum_{l\leq j} X_l^{(n)}\right) < (j-l+1)\cdot C\right)$$
$$\wedge P\left(\sum_{i\leq l\leq i+k+1} X_j^{(n)} \leq (k+1)\cdot C\right).$$

Unfortunately, we cannot directly bound these values for $X_i^{(n)}$ or even for $Y_i^{(n)}$ as we cannot estimate the probability that the AD counter of a bucket is 0. However, since for each k we are looking at an intersection of probabilities, we can simply bound the expectation using only a single item of the intersection. In particular, we can estimate $P(LS(X_i^{(n)})=2)$ with the term $P((X_i^{(n)}+X_{i+1}^{(n)}) \geq 2\cdot C)$ and $P(LS(X_i^{(n)})=3)$ with the term $P((X_i^{(n)}+X_{i+1}^{(n)}+X_{i+2}^{(n)}) \geq 3\cdot C)$.

We can therefore bound the expectation of LS that is by definition:

$$E\left(LS\left(X_i^{(n)}\right)\right) \leq \sum_{k=1}^{n} k\cdot P\left(LS\left(X_i^{(n)}\right)=k\right)$$

to be:

$$E\left(LS\left(X_i^{(n)}\right)\right) \leq \sum_{k=1}^{n} k\cdot P\left(\left(\sum_{j=i}^{j=i+k} X_j^{(n)}\right) < k\cdot C\right).$$

Now, since $X_i^{(n)}$s are not independent, we still cannot evaluate this expectation. However, if we switch $X_i^{(n)}$ with the corresponding independent variables $Y_i^{(n)}$, we can calculate the expectation with the Poisson tail function. That is:

$$E\left(LS\left(Y_i^{(N)}\right)\right) \leq \sum_{k=1}^{n} k\cdot \text{PoissonTail}\left(k\cdot\frac{N}{B}, k\cdot C\right)$$

where $PoissonTail(Z,W)$ is the tail function of the Poisson distribution, which describes the probability for a Poisson variable with expectancy Z to be larger than W. We note

that the above is true since a sum of Poisson variables is also a Poisson random variable (with matching expectancy).

Having bounded $E(LS)$, we can now simply bound $E(SIZE(Y_1,...,Y_N)))$ with the following calculation:

$$E\left(SIZE\left(Y_1^{(n)},...,Y_B^{(n)}\right)\right) = E\left(\frac{\sum_{i=1}^{B} LS\left(Y_i^{(n)}\right)}{B}\right)$$

$$= \frac{B \cdot E\Big(LS\left(Y_1^{(n)}\right)\Big)}{B} = E\Big(LS\left(Y_1^{(n)}\right)\Big).$$

Activating the lemma, we get:

$$E\left(SIZE\left(X_1^{(n)},...,X_B^{(n)}\right)\right) \leq 2E\left(SIZE\left(Y_1^{(n)},...,Y_B^{(n)}\right)\right)$$

$$\leq 2E\Big(LS\left(Y_1^{(n)}\right)\Big)$$

$$\leq 2\Bigg(\sum_{k=1}^{n} k \cdot \text{PoissonTail}\left(k \cdot \frac{N}{B}, k \cdot C\right)\Bigg).$$

We showed an upper bound on the number of buckets that are effected on average per population as a function of the load. We now show how to translate this bound to an estimation of the operation complexity in terms of shifted cells.

To this end, we require the following assumption: An operation that shifts cells shifts on average half of the cells in the logical bucket. Under this assumption, if $2E\Big(LS\left(Y_1^{(n)}\right)\Big)$ buckets are shifted, and each bucket is expected to have N/B items, we estimate the complexity in cells (fingerprints) to be:

$$\left(X_1^{(n)},...,X_B^{(n)}\right)\frac{N}{B}\cdot\frac{1}{2} \leq$$

$$2E\left(LS\left(Y^{(n)}\right)\right)\cdot\frac{N}{B}\cdot\frac{1}{2} = E\left(LS\left(Y^{(n)}\right)\right)\cdot\frac{N}{B}.$$

B. False Positive in Multisets

It is well known that when all fingerprints are of the same size and the load factor is λ, $\lceil\frac{\lambda}{\epsilon}\rceil$ bits are required in order to provide a false positive of ϵ. When we collect statistics, the notion of λ is not sufficient to determine the false positive rate. The reason is that although counter cells take room in the bucket, they are never compared to fingerprints. We therefore introduce the notion of *fingerprint load factor* (λ_{fp}). That is, the average amount of fingerprint cells in a chain. In that case, $\lceil\frac{\lambda_{fp}}{\epsilon}\rceil$ bits are required in order to provide a false positive rate of ϵ.

Denote NU the number of unique items in the workload and NB the number of buckets. The fingerprint load factor is defined in the following way: $\lambda_{fp} = \frac{NU}{NB \cdot L}$. We note that similarly to the corresponding set problem, in this case fingerprints are assigned randomly to buckets.

C. Mapping Error

When multisets are involved, we look at two different metrics for the quality of the approximation. The first is false positive, which measures the probability that an item that did not appear in the sample is evaluated as an item that did appear in the sample. However, false positives do not give the complete picture, because if the item did appear in the sample, its approximated value can actually be higher than its real value.

The second error is caused when two different items hash to the same bucket and chain and also have the same fingerprint. In that case, the estimation of both counters contains an error. We therefore define this error as a mapping error.

Mapping Error Denote *MappingError* (*ME*) the probability that for a contained item T, the approximated value is different than the real value:

$$MappingError = \\ P\left(ApproximateValue\left(T\right) \neq RealValue\left(T\right) \mid \left(T \in S\right)\right)$$

In order to bound the mapping error, we first calculate the probability that two or more fingerprints are identical for a given chain size.

Definition Let Z_i be a random variable s.t. $Z_i = 1$ if out of a random set of i fingerprints, at least two fingerprints have the same value. Z_i is 0 otherwise.

Clearly, Z_1 is always 0. $P(Z_2 = 1)$ is simply the probability that two fingerprints are identical, $\frac{1}{2^S}$. In general, Z_i is the complement of the case where all the i fingerprints are different and is therefore: $\left(1-\left(1-\frac{1}{2^S}\right)^i\right)$.

Definition Denote W_i a random variable that represents the number of fingerprints stored in chain i. That is, $W_i = k$ if there are k different fingerprints stored in chain i.

Since λ_{fp} is very small compared to the number of fingerprints stored in the table, we assume that $W_i \sim Poission\left(\lambda_{fp}\right)$. In order to give an upper bound on the approximation error, if at least two items that hash to a chain have identical fingerprints, we assume that *all* items that hash to that chain have identical fingerprints, resulting in k wrong counters. In reality, usually only two fingerprints collide. However, since the common case that constitutes the vast majority of the error is the case where the chain has only two fingerprints, this assumption is reasonable. Hence, the mapping error can be bound by:

$$MappingError \leq \frac{\sum_i P(W_i=k)\wedge(Z_k=1)\cdot k}{L} \\ = \sum_i P\left(W_i = k\right) \wedge \left(Z_k = 1\right) \cdot k \\ = \sum_{i=k} Poission\left(\lambda_{fp}, k\right) \cdot \left(1-\left(1-\frac{1}{2^S}\right)^k\right) \cdot k$$

D. Required Space

For the dynamic set problem, it is easy to see that the table is required to store a single fingerprint per item. In principle, the same observation holds for the multiset problem as well. That is, in the worst case every item appears only once and therefore every addition requires an additional fingerprint. Therefore,

without knowing anything about the data, no better bound can be given to the problem.

In order to bound the amount of space required by a table for a given workload, we start by analyzing the amount of table cells required to store a single item and its value if its final value is known in advance.

1) Space Per Item Analysis:

Definition Denote $V(T)$ the value of an item T and $Cost(T)$ the amount of table cells that are needed to store T and $V(T)$ in TinyTable.

Since the item is coded in a chain, with a fingerprint cell followed by counter parts until all bits of the counter are present, the required number of cells to store an item T with a value $V(T)$ is: $Cost\,(T) = 1 + \lceil \log_{2^S}(V\,(T) - 1) \rceil$ cells in order to store the item and its value.

If we chose to use the suggested optimization, we can do slightly better for items that are first in their chain. That is, since the first cell in a chain always stores a fingerprint, we gain a single bit counter for first in their chains items. The cell cost for such items will therefore be: $Cost\,(T) = 1 + \left\lceil \log_{2^S}\left(\frac{V(T)}{2} - 1\right)\right\rceil$.

2) Total Space Analysis: In order to bound the total number of cells required by TinyTable, similarly to [26], we assume that we know two parameters that characterize the data: NU - the number of unique items that is also required in order to evaluate the false positive probability, and the average item value m.

Denote $l = \lceil \log(m-1) \rceil \bmod S$, i.e., l is the number of extra bits allocated in the table for the average item beyond the minimal required to count to $m-1$ (the fingerprint itself already represents the first occurrence of the item). We claim that the total number of cells in the table required to accommodate all items and their values is upper bounded by

$$\left(Cost(m) + \frac{1}{2^l}\right) \cdot NU.$$

Intuitively, given that m is the average, the above bound holds since the portion of items requiring more than the extra allocated l bits can be at most $\frac{1}{2^l}$. Any item requiring more than $Cost(m)+1$ cells in the table (if exists) implies that there are fewer items requiring $Cost(m)+1$ cells, and these are canceling each other out.

V. Results

A. Overview

In this section, we evaluate both our analysis and our Java implementation of TinyTable [3]. The evaluation starts with understanding the tradeoff and optimization space of TinyTable. The main space efficiency and update complexity tradeoff is evaluated in Section V-B. Section V-C explains how to properly configure anchor distance counters. Section V-D evaluates the space/accuracy tradeoff of TinyTable compared to other counting Bloom filter suggestions. This is followed comparing the space/accuracy tradeoff of TinyTable to other approximate counting suggestions in SectionsV-E and V-F.

B. Addition/Removal Complexity

In TinyTable, read operations only require offset calculation and can take advantage of efficient bitwise instructions. Read complexity is therefore constant regardless of the load. However, removals and additions require shifting fingerprints to keep the fingerprint array synchronized with the index. The complexity of these operations depends on the load placed upon the table. In particular, the denser the table is the more complex these operations are.

We evaluated the effect of table load on the complexity of add/remove operations under different table and bucket sizes. Figure 7(a) describes the progression of the add/remove complexity as tables of different sizes become increasingly loaded. The basic configuration used is 40 allocated cells per bucket, and the tested table sizes are 100k buckets, 10k buckets and 1k buckets. Since the lines are almost identical, we conclude that table size plays a minor role in the complexity of operations.

This figure also evaluates our complexity analysis. As can be observed, for most values of α, it is accurate up to a factor of 2, providing a good upper estimation to the operation complexity.

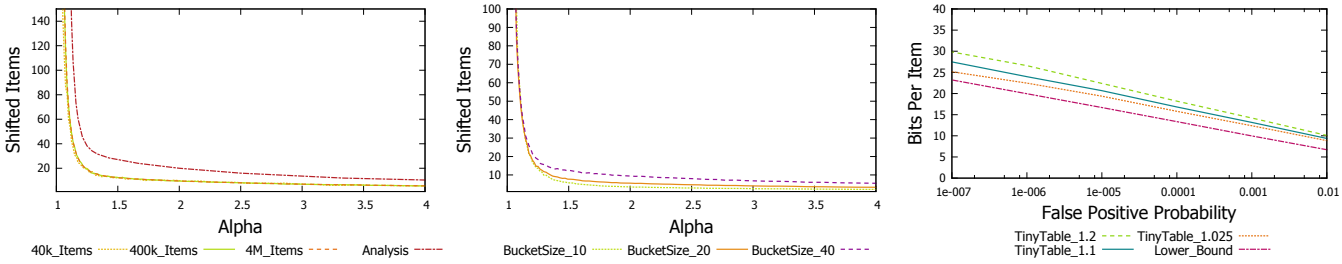
Figure 7(b) exhibits the same experiment for a table with a capacity of 4 million items. This time, however, we test different bucket sizes. That is, we tested a table with 400k buckets of 10 cells, 200k buckets of 20 cells and 100k buckets of 40 cells. As can be observed, the bucket size impacts the operation complexity when the table is lightly loaded. However, since we seek good space/accuracy tradeoff, lightly loaded configurations are not very interesting in our context. In dense configurations ($\alpha <\approx 1.15$), the operation complexity is dominated by the table load rather than the bucket size.

For the rest of the evaluation we focus on three configurations that are attractive for different problems. The only difference between these configurations is the parameter α. In particular, all configurations are expected to contain 40 items per bucket ($N/B = 40$), but differ in the number of allocated items per bucket (C), which determines α and controls their operation complexity.

The first configuration is designed for situations where reads are dominant and updates are rare. In this configuration, $C = 41$ and therefore $\alpha = 1.025$. Counting Bloom filters with similar update complexity were found useful for some applications [30], [31]. This configuration considerably improves their space efficiency and read performance, while keeping a similar update complexity.

In the second configuration, $C = 44$ and therefore $\alpha = 1.1$. This configuration is good for general purpose applications as it offers a similar update speed compared to Bloom filters, with a significantly faster query speed. It is more space efficient than a Bloom filter, and also supports removal capability and is therefore suggested in places where reasonable performance is required and space is a bottleneck. We believe that this is the case with the majority of Bloom filter applications.

In the last configuration, $C = 48$ and therefore $\alpha = 1.2$. This configuration is significantly faster than the previous one but requires slightly more space. Its space consumption is still



(a) Different table sizes (b) Different bucket sizes (c) Space/accuracy for different table loads

Fig. 7. Effect of table load on addition/removal complexity

Time To:	Query			Add/Remove			Read Complexity	Cells Shifted	Cache Lines
Configuration/False Positive Probability	1%	0.1%	0.01%	1%	0.1%	0.01%		Add/Remove	(aligned)
TinyTable, $\alpha = 1.2$	0.06	0.06	0.06	0.28	0.28	0.28	O(1)	20 cells	1
TinyTable, $\alpha = 1.1$	0.06	0.06	0.06	0.61	0.62	0.64	O(1)	55 cells	1-2
TinyTable, $\alpha = 1.025$	0.06	0.07	0.07	7.39	7.52	10.46	O(1)	1500 cells	18-36
Bloom Filter	0.68	0.97	1.34	0.65	0.96	1.33	$O(\frac{1}{\epsilon})$	N.A	7-13

TABLE I
TIME TO PERFORM 1 MILLION OPERATIONS (SECONDS) AND OPERATION COMPLEXITY FOR DIFFERENT TINYTABLE CONFIGURATIONS.

relatively low and it is especially attractive for problems where the update performance is a bottleneck of the system. This configuration is also slightly smaller than Bloom filters for relatively low false positive rates.

The complexity and operation speed of each configuration for different false positive rates is summarized in Table I. For this measurement we created tables from the three configurations and filled each with 1 million items. We then measured the time required to perform 1 million (positive) queries and add/remove operations. We also measured the average operation complexity in terms of shifted cells and minimal number of x86 cache lines that can contain that amount of shifted cells. For reference, we also tested an open source Bloom filter implementation[2]. All measurements are performed on the same Intel i7@3.2GHZ cpu, and the data structures were contained entirely in memory.

As can be observed, query time is fast for all configurations as it is implemented with efficient bitwise operations. Add and remove operations are significantly slower, yet the $\alpha = 1.1$ and $\alpha = 1.2$ configurations are also faster than the Bloom filter. As expected, the $\alpha = 1.025$ configuration is very slow for add/remove operations. Its main benefit is the fast query time that is hardly affected by α.

C. Sizing Anchor Distance Counters

Table II describes the relationship between the number of bits allocated to anchor distance counters and their overflow probability. For the rest of our measurements, we use 5 bits anchor counters for the $\alpha = 1.1$ and $\alpha = 1.2$ configurations. At this point, the probability that the counter is sufficient is $\approx 99.86\%$ in the 1.1 configuration, and higher than 99.999% in the 1.2 configuration. The 1.025 configuration was evaluated with 12 bits anchor distance counters that are enough over 99.999% of the cases.

[2]The project can be found at https://code.google.com/p/java-bloomfilter/

	Probability Not to Overflow			
Configuration	50%	95%	99%	99.9%
$\alpha = 1.2$	1	3	4	4
$\alpha = 1.1$	2	4	5	6
$\alpha = 1.025$	5	8	9	10

TABLE II
REQUIRED AMOUNT OF BITS IN ANCHOR DISTANCE COUNTERS

In our experiments, the anchor recovery process is therefore a very rare event. Even when the recovery process is initiated, the previous bucket whose anchor did not overflow is typically just the previous bucket, as over 99% of the anchors do not overflow in each of our configurations.

D. Counting Bloom Filter Functionality

In order to complete the picture of table load vs. speed tradeoff, we evaluate the space/accuracy tradeoff of the different TinyTable configurations. Figure 7(c) describes the three different configurations ($\alpha = 1.2$, $\alpha = 1.1$ and $\alpha = 1.025\%$) compared to the theoretical lower bound [9]. As can be observed, TinyTable is fairly efficient compared to the lower bound. Further, TinyTable 1.025 has only a minor ($\approx 2.5\%$) multiplicative factor from the lower bound. The distance from the lower bound ≈ 2.3 bits is mainly explained by the indexing overheads (2 bits when $\alpha = 1$).

Next, we compare the more practical 1.2 and 1.1 configurations with the state of art alternatives[3]. Figure 8(a) evaluates the space/accuracy tradeoff of these alternatives compared to rank indexed hashing and d-left hashing, two hash table based solutions that are considered state of the art for the problem. These constructions are configured according to their corresponding authors' instructions. As can be observed,

[3]Recall that $\alpha = 1.025$ is only useful when updates are extremely rare.

False Positive	TinyTable, $\alpha = 1.1$	Rank CBF	d-left CBF	VI-CBF	Bloom Filter (No removals)	Comparison vs Rank	vs d-left	vs VI-CBF	vs Bloom Filter
1%	9.4	13	17.6	25	9.6	-28%	-47%	-64%	-2%
0.1%	13.2	16.8	22.3	37.8	14.4	-21%	-41%	-65%	-8%
0.01%	16.8	20.6	26.4	50	19.1	-18%	-36%	-66%	-12%

TABLE III
REQUIRED SPACE (IN BITS) PER ELEMENT FOR THE SAME FALSE POSITIVE RATE (WITH REMOVALS) - NOTE THAT BLOOM FILTERS DO NOT SUPPORT REMOVALS

False Positive	$\alpha = 1.2$	$\alpha = 1.1$	$\alpha = 1.025$	Rank	Lower Bound
1%	10.1	9.4	9.1	13	6.7
0.1%	14.2	13.2	12.6	16.8	10
0.01%	18.2	16.8	16	20.6	13.4

TABLE IV
SPACE CONSUMPTION OF DIFFERENT TINYTABLE CONFIGURATIONS COMPARED TO RANK INDEXED HASHING AND THE THEORETICAL LOWER BOUND

both TinyTable configurations are more space efficient than previously suggested constructions.

Table III gives an additional perspective. In this table, we also compare TinyTable to a state of the art Bloom filter based CBF construction called *variable increment CBF (VI-CBF)* [36] and to a standard Bloom filter that does not support removals, but is considered very space efficient. We conclude that TinyTable $\alpha = 1.1$ consumes $28-18\%$ less space than the most efficient alternative for the range. TinyTable even consumes $2-12\%$ less space than a (plain) Bloom filter that does not support removals. This fact is especially significant, since most Bloom filter based CBF variants consume significantly more space than a regular Bloom filter.

Table IV completes our evaluation by comparing the different TinyTable configurations we suggest to both the state of the art and to the theoretical lower bound (for the case of perfect hashing). We note that even TinyTable 1.2 offers a better space/accuracy tradeoff than the state of the art while keeping average add/remove complexity very low. For this range, it consumes $22 - 12\%$ less space than rank indexed hashing. Further, TinyTable 1.025 is very close to the theoretical lower bound. While its additions and removals are slow, reading it is still very efficient. Constructions with similar limitations were also suggested for several applications [30], [31]. Since these constructions consume more space than a plain Bloom filter, TinyTable 1.025 is significantly more space efficient than they are.

E. Approximate Counting Functionality

We compare TinyTable to two previously suggested space efficient approximate multiset algorithms. The first one is *adaptive Bloom filter (a.k.a. ABF)* [33], which is considered very space efficient, but is very limited in its applicability since it does not support negative/variable updates and the number of hash functions required to read/update a value scales linearly with the value. The second one is the well known spectral Bloom filter (a.k.a. SBF) [12], which is a Bloom filter based construction with an efficient counter encoding that was shown to have applications in a wide variety of domains. Both are configured according to their respective authors' instructions. Table V gives a high level overview of key properties of the protocols. As can be seen, both TinyTable and spectral Bloom filters support variable increments and removals. The main advantage of TinyTable is the fact that it only requires a single hash function and it accesses memory in add/remove operations in a serial manner.

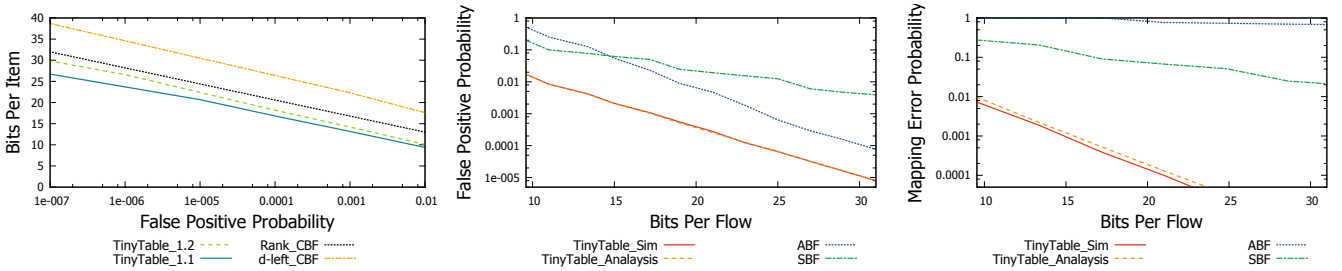
Flow Given a stream of data, a *flow* consists of all the occurrences of the same item within the stream.

Our first experiment is a short measurement of 1.25 Million continuous packets taken from [22], consisting of 125k different flows, where each flow is identified by the combination of source IP address, source port, destination IP address, and destination port. Our goal is to count the number of packets in each flow. In this (and subsequent) measurements, we configured TinyTable for the more spacious 1.2 configuration. That is, we analytically configured it according to our bound so that the table has at least 20% free space on average. We justify the use of the more specious configuration as realistic workloads often do not have 100% certainty about the actual load. This configuration is able to handle a slightly larger actual load than anticipated and still remain within an acceptable performance envelope.

Figures 8(b) and 8(c) illustrate both the analytical and simulated behavior of TinyTable in comparison with ABF and SBF. As can be seen, our false positive analysis is accurate while the mapping error simplified analysis is indeed an upper bound. Further, TinyTable significantly outperforms both SBF and ABF in both false positive and mapping error.

Table VI summarizes the number of bits per flow required to achieve the desired false positive or mapping error rate. In order to understand how effective our variable counter length optimization is, we also compared ourself to a fixed counter sized version of a 1.2 TinyTable called *SimpleTable*. This table does not use our variable counter length optimization. Instead, each fingerprint is attached a 21 bits counter in order to make sure that it is able to represent the maximal value that may appear in the workload. As can be observed, even SimpleTable consumes less space than an SBF, especially for low false positive/mapping error rates. This is despite the fact that the SBF uses an efficient coding technique for its counters. The reason is that compact hash tables in general are inheritably more suitable for the approximate counting problem.

Further, TinyTable achieves a better space/accuracy tradeoff than the alternatives. It is significantly better than SimpleTable, as it is able to associate counters to fingerprints on the fly.



(a) CBF functionality (workload independent) [22] (b) Counting functionality, false positives in [22] (c) Counting functionality, mapping error in [22]

Fig. 8. Accuracy of TinyTable for both CBF and counting functionality

Protocol	Hash Functions	Memory Access Pattern	Memory Accesses Per Read	Variable Updates	Negative Updates
TinyTable	1	Serial	LOG(VALUE)	Yes	Yes
SBF	ε	Random Access	ε	Yes	Yes
ABF	$\varepsilon + VALUE$	Random Access	$VALUE$	No	No

TABLE V
COMPARISON OF SELECTED FEATURES OF TINYTABLE, SBF AND ABF

False Positive / Map Error	TinyTable	SimpleTable	ABF	SBF	Comparison		
					vs Simple	vs ABF	vs SBF
1%	10.6/9	35.7/35.25	25.6/100+	48/34.5	-70%/-60%	-59%/N.A	-78%/-69%
0.1%	18.8/15.8	39.9/36.7	33.2/100+	72/51	-53%/-75%	-43%/N.A	-85%/-79%
0.01%	23.8/17.6	43.1/39.6	40/100+	95.5/72	-40%/-55%	-41%/N.A	-88%/-85%

TABLE VI
ACCURACY/SPACE TRADEOFF (FALSE POSITIVE/MAPPING ERROR). SHORT MEASUREMENT FROM [22]

Unfortunately, since our design is constrained to use the same sized fingerprints and counter parts, the benefit is lesser for lower false positive rates. The reason for this is that low false positive rates require longer fingerprints, and therefore the counter allocation is done using larger parts (cells) and is therefore less efficient. Yet, even at 0.01% false positive rate, TinyTable consumes 40% less space than SimpleTable.

To complete the picture, notice that for both SBF and TinyTable, the mapping error is significantly smaller than the false positive error. But this is not always the case. In particular, since ABF uses the same bitmap for both containment and for counting, the estimation of flows often indicates a value higher than the real stored value (a mapping error).

F. What Happens When Counters Grow?

Although TinyTable is quite efficient for small counter values, TinyTable fully supports variable increments and can therefore also efficiently count per flow traffic. Table VII summarizes the space/accuracy tradeoff offered by TinyTable 1.2 on various real life TCP packet traces. As can be observed, TinyTable is also efficient when the average flow is quite large. Further, TinyTable scales well with the false positive rate. In particular, if we consider that just an explicit representation of an IPV4 flow (even without a counter) requires 128 bits, approximate representation is an attractive way to go.

Trace	Unique Flows	Combined Traffic (bytes)	Bits Per Flow		
			1%	0.1%	0.01%
[22]	1,107,541	19,227,856,695	41.3	44.7	51.8
[23]	949,846	16,561,081,158	41.3	44.7	51.9
[24]	3,067,001	13,259,440,602	40.3	44.4	46.2
[25]	1,295,103	32,913,196,949	41.3	44.7	51.4

TABLE VII
SPACE/ACCURACY TRADEOFF OF TINYTABLE WITH $\alpha = 1.2$ WHEN RECORDING PER FLOW TRAFFIC

VI. CONCLUSIONS

In this work, we have introduced TinyTable, a compact fingerprint hash table that offers an alternative to both counting Bloom filters and spectral Bloom filters since it explicitly supports removals as well as variable value increments and decrements. TinyTable combines features that resemble both a linear probing hash table and a chain based hash table.

Read operations are performed by very efficient bitwise operations and are therefore very fast. Update complexity is somewhat similar to that of a linear probing hash table as updates become more complex as the table load increases.

TinyTable can use all the allocated memory without overflowing and therefore offers a good space efficiency to update speed tradeoff.

TinyTable allows different applications to use it with different settings according to the actual requirements of the workload. TinyTable is slightly smaller than a Bloom filter for practical false positive rates, making it more space efficient than Bloom filter based counting Bloom filter suggestions. It is also more space efficient than previously suggested hash table constructions.

Additionally, TinyTable efficiently supports statistics and

multiset representation. It extends the utility of previously suggested rank indexing techniques in order to associate each fingerprint with a variable sized counter. To the best of our knowledge, TinyTable is the first fingerprint hash table solution that fully supports multisets. We show that TinyTable is considerably more space efficient than SBF, ABF and a fixed counter length fingerprint hash table.

Finally, we have shown that TinyTable can efficiently count the total traffic per flow for several real life Internet traces with a moderate per flow memory. TinyTable is an actively developed Java-based open source project available at [3].

REFERENCES

[1] http://blog.alexyakunin.com/2010/03/nice-bloom-filter-application.html.

[2] squid-cache.org. http://www.squid-cache.org/.

[3] TinyTable: A Java based implementation. https://code.google.com/p/tinytable/.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[5] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. In *SIGCOMM*, pages 315–326, 2006.

[6] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Bloom filters via d-left hashing and dynamic bit reassignment. In *Proc. of the Allerton Conf. on Communication, Control and Computing*, 2006.

[7] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *14th Annual European Symposium on Algorithms, LNCS 4168*, pages 684–695, 2006.

[8] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[9] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman. Exact and approximate membership testers. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 59–65, New York, NY, USA, 1978. ACM.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[11] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: An efficient data structure for static support lookup tables. In *Proc. of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[12] S. Cohen and Y. Matias. Spectral bloom filters. In *Proc. of the 2003 ACM SIGMOD International Conf. on Management of Data*, SIGMOD, pages 241–252, New York, NY, USA, 2003. ACM.

[13] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55:29–38, 2004.

[14] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM, pages 201–212, New York, NY, USA, 2003. ACM.

[15] G. Einziger and R. Friedman. Postman: An elastic highly resilient publish/subscribe framework for self sustained service independent P2P networks. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2014.

[16] G. Einziger and R. Friedman. TinyLFU: A highly efficient cache admission policy. In *Euromicro PDP*, 2014.

[17] G. Einziger and R. Friedman. TinySet - an access efficient self adjusting bloom filter construction. Technical Report CS-2015-03, Computer Science Department, Technion, March 2014.

[18] G. Einziger, R. Friedman, and Y. Kantor. Shades: Expediting Kademlia's lookup process. In *Euro-Par*, 2014.

[19] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *SIGCOMM Comput. Commun. Rev.*, 32(4):323–336, Aug. 2002.

[20] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

[21] D. Ficara, A. D. Pietro, S. Giordano, G. Procissi, and F. Vitucci. Enhancing counting bloom filters through huffman-coded multilayer structures. *IEEE/ACM Trans. Netw.*, 18(6):1977–1987, 2010.

[22] P. Hick. CAIDA Anonymized 2008 Internet Trace, equinix-chicago 2008-03-19 19:00-20:00 UTC, Direction A. http://www.caida.org/data/monitors/passive-equinix-chicago.xml.

[23] P. Hick. CAIDA Anonymized 2008 Internet Trace, equinix-chicago 2008-03-19 19:00-20:00 UTC, Direction B. http://www.caida.org/data/monitors/passive-equinix-chicago.xml.

[24] P. Hick. CAIDA Anonymized 2013 Internet Trace, equinix-sanjose 2013-1-17 13:55 UTC, Direction B. http://www.caida.org/data/monitors/passive-equinix-sanjose.xml.

[25] P. Hick. CAIDA Anonymized 2014 Internet Trace, equinix-chicago 2014-03-20 13:55 UTC, Direction B. http://www.caida.org/data/monitors/passive-equinix-chicago.xml.

[26] N. Hua, B. Lin, J. J. Xu, and H. C. Zhao. Brick: A novel exact active statistics counter architecture. In *Proc. of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 89–98, New York, NY, USA, 2008. ACM.

[27] N. Hua, H. C. Zhao, B. Lin, and J. Xu. Rank-indexed hashing: A compact construction of bloom filters and variants. In *ICNP*, pages 73–82. IEEE, 2008.

[28] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[29] T. Lee, K. Kim, and H.-J. Kim. Join processing using bloom filter in mapreduce. In *Proc. of the 2012 ACM Research in Applied Computation Symposium*, RACS '12, pages 100–105, New York, NY, USA, 2012. ACM.

[30] L. Li, B. Wang, and J. Lan. A variable length counting bloom filter. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conf. on*, volume 3, pages V3–504–V3–508, 2010.

[31] W. Li, K. Huang, D. Zhang, and Z. Qin. Accurate counting bloom filters for large-scale data processing. *Mathematical Problems in Engineering*, 2013, 2013.

[32] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: a novel counter architecture for per-flow measurement. In *Proc. of the ACM SIGMETRICS Int. Conf. on Measurement and modeling of computer systems*, pages 121–132. ACM, 2008.

[33] Y. Matsumoto, H. Hazeyama, and Y. Kadobayashi. Adaptive bloom filter: A space-efficient counting algorithm for unpredictable network traffic. *IEICE - Trans. Inf. Syst.*, E91-D(5):1292–1299, May 2008.

[34] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[35] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proc. of the 1st USENIX Conf. on File and Storage Technologies*, FAST, Berkeley, CA, USA, 2002. USENIX Association.

[36] O. Rottenstreich, Y. Kanizo, and I. Keslassy. The variable-increment counting bloom filter. In *INFOCOM*, pages 1880–1888, 2012.

[37] H. Song, F. Hao, M. S. Kodialam, and T. V. Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *INFOCOM*, pages 2518–2526. IEEE, 2009.