TinySet - An Access Efficient Self Adjusting Bloom Filter Construction[*]

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

Abstract

Bloom filters are a very popular and efficient data structure for approximate set membership queries. However, Bloom filters have several key limitations as they require 44% more space than the lower bound, their operations access multiple memory words and they do not support removals.

This work presents TinySet, an alternative Bloom filter construction that is more space efficient than Bloom filters for false positive rates smaller than 2.8%, accesses only a single memory word and partially supports removals. TinySet is mathematically analyzed and extensively tested and is shown to be fast and more space efficient than a variety of Bloom filter variants. TinySet also has low sensitivity to configuration parameters and is therefore more flexible than a Bloom filter.

1 Introduction

Approximate set membership data structures offer a memory and computation efficient set representation. They obtain this efficiency by trading exact answers to membership queries with approximate results. In particular, negative answers by these data structures are always correct, while positive answers are correct with a probability of $1-\varepsilon$, where ε is a performance parameter; the smaller ε is the larger the data structure is. These structures are attractive when the available space is limited.

The most popular example of such structures is Bloom filter [8]. Bloom filters (and variants) are extensively used in network caching [21, 42], routing and prefix matching [15, 43], security [7, 9, 23] and many more [11].

Bloom filters are also implemented in some very successful widely deployed systems. For example, Mellanox's IB Switch System [4] uses Bloom filters in order to provide monitoring capability. Google's Big Table [13] and Apache Cassandra [31] employ them in order to avoid performing disk lookups for non existing data. In these systems, Bloom filters are stored in main memory and their content approximates that of a significantly larger disk. If the data is stored on disk, we are guaranteed a positive reply from the Bloom filter, and therefore never miss the disc content. False positives cause redundant accesses to the disk. Yet, as the false positive ratio is relatively low, these are acceptable. The main benefit comes from a negative answer, as these answers are always correct. That is, when the Bloom filter gives a negative result, these systems avoid a disk access entirely.

Using a similar trick, Bloom filters are also suggested in order to reduce communication in MapReduce [32] with a variant of a traditional Bloom join protocol [35], initially suggested only for databases. Google Chrome [1] uses a Bloom filter in order to maintain an approximated list of malicious web sites. Google servers are only contacted for sites whose URL is contained in the Bloom filter. In that case, the exact list is checked and a definite response is returned to the user.

[*]This research was partially funded by the Israeli Ministry of Science Technology and Space grant #3-11023 and the Technion Hasso Platner Research School.

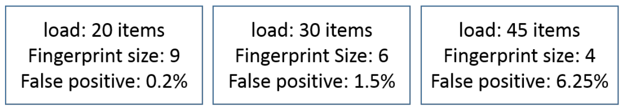


Figure 1: A high level overview of TinySet. The structure is partitioned into many fixed size blocks. Each block is dynamically configured according to the actual load placed on it.

BitCoin [37], a very successful peer-to-peer currency, uses Bloom filters in order to expedite transaction verification [2]. In addition, Bloom filters are used for cache management [18, 20] as well as distributed cache architectures. In this domain, a caching service publishes an approximation of the cache content, which is significantly smaller than an accurate representation of the cache content. Famous examples include SummeryCache [21] and Squid [5].

Bloom filters are also used in distributed routing. For example, OceanStore [30], a distributed storage system, uses a cluster of Bloom filters called *Attenuated Bloom Filters* in order to route requests to their destination. Further, routing using Bloom filters was suggested in the context of publish/subscribe [27, 28, 17] where Bloom filters determine if matching subscribers may exist in a certain direction.

Still, despite of their enormous popularity and success, Bloom filters have several key limitations. First, they require $\approx 44\%$ more space than the theoretical lower bound [12]. Second, their number of required hash functions is proportional to ε, and each hash function calculation is followed by a memory access. Finally, Bloom filters do not support remove operations.

1.1 Our Contribution

In this paper, we introduce TinySet, a novel data structure for approximate membership queries. TinySet combines flavors from both compact hash tables [10, 25] and blocked Bloom filters [39, 40] in order to provide a combination of good properties, all in a single data structure.

In particular, TinySet is access efficient as its operations are limited to a single block of fixed size memory space. For example, this block size can be tailored to a CPU cache line. In that case, an operation only touches a single cache line and the performance significantly improves. Fixed memory complexity is also an attractive feature for hardware implementations. We show that although TinySet is a more sophisticated algorithm, it achieves better performance than Bloom filters, especially for query operations.

In addition, TinySet is more flexible than Bloom filters as it dynamically changes its configuration to suit the actual load. Unlike Bloom filters, TinySet also supports removals, yet these gradually degrade its space efficiency over time. Finally, many TinySet configurations are more space efficient than Bloom filters. The most space efficient configuration that is presented in this paper is smaller than Bloom filters for false positive rates lower than 2.8%.

A high level intuition about TinySet is given in Figure 1. In this example, 3 independent blocks are drawn. Each block possibly uses a slightly different encoding method. An arriving item is hashed and inserted to one of these blocks. Prior to the insertion, the block reconfigures itself in order to represent the stored items as accurately as possible. In this case, the first block is under loaded, as it contains a lower than average number of items. It can thus achieve a significantly lower false positive rate. The second block is a typical block that contains an average number of items and achieves an average false positive rate. Finally, the third one is overloaded and contains more items than initially intended. It therefore yields a higher false positive rate. The crux is that unlike previous suggestions, TinySet efficiently utilizes a different configuration for each individual block. The overall false positive rate is an average of many blocks with varying accuracy, and extreme loads are rare.

We also present a mathematical analysis of TinySet. This is complemented by an extensive empirical performance study demonstrating and exploring TinySet's behavior under various conditions and comparing TinySet with several previously proposed alternatives including Bloom filters [8], d-left hashing [10], rank indexed hashing [25], blocked Bloom filters [40] and balanced Bloom filters [40].

Paper Roadmap An overview of background and related work is described in Section 2. TinySet is presented in Section 3 followed by an analysis in Section 4. A comprehensive performance study of TinySet including a comparison with Bloom filters, d-left hashing, rank indexed hashing, blocked Bloom filters and balanced Bloom filters appears in Section 5. We conclude with a discussion in Section 6.

2 Background and Related Work

2.1 Bloom Filters Variants

Over the years, many data structures were suggested in order to improve different aspects of Bloom filters. For example, *compressed Bloom filters* [36] use compression techniques in order to achieve optimal space efficiency at the expense of calculation speed.

Blocked Bloom filter (BlockedBF) [39, 40] partitions a Bloom filter into many fixed sized blocks, each containing an independent Bloom filter. An arriving item is first hashed to a block and is then inserted to the Bloom filter of that block.

Since blocked Bloom filters only access a single block, they utilize the memory more efficiently and typically achieve higher throughput and consume less power [29]. Unfortunately, the unequal load placed on each block makes this suggestion less space efficient than bloom filters.

Alternatively, *Balanced Bloom filters (BalancedBF) [29]* improve space efficiency with load balancing techniques. That is, a small fraction of the items are not inserted to the blocked Bloom filter and are separately maintained in an *overflow list* that is implemented with TCAM memory. However, even when employing these techniques, BalancedBF is still less space efficient than a Bloom filter. Moreover, this technique can similarly augment the accuracy of TinySet that also benefits from a more balanced load.

Counting Bloom filters (CBF) [30] enhance Bloom filters in order to support removals. Alas, these are significantly less space efficient when compared to standard Bloom filters even in sophisticated, state of the art, implementations [22, 26, 33, 41]. Despite their space inefficiency, removal functionality is essential to many problems and counting Bloom filters are therefore extensively used. An interesting kind of a counting Bloom filter is the *Inverted Bloom filter* [24] that can also associate values with the stored items.

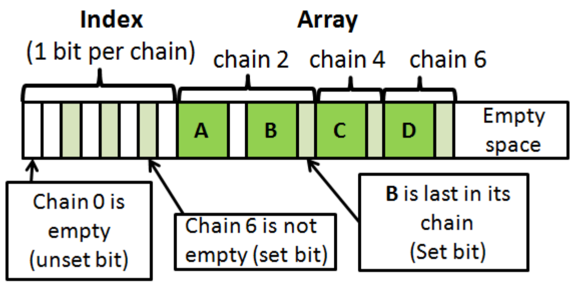
On a more theoretical note, a space optimal counting Bloom filter is suggested in [38]. This approach is based on compact hash tables and sophisticated encoding. Although it is asymptotically optimal, it is not very attractive in practice due to its high overheads. The authors do suggest a practical variant that does not support removals or access memory efficiently.

2.2 Hash Table Based Bloom Filters

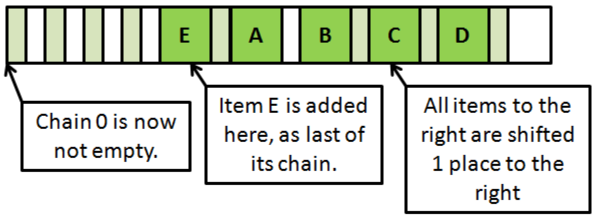
In principle, a hash table can be used as an alternative for Bloom filters and their variants. In particular, it is well known that when the set of items is static and known in advance, we can calculate a *perfect hash function*. Since this function hashes all the items in the set without collisions, a simple array can be used to store fingerprints of the items in the set [11].

More formally, a perfect hash function $P : S \to [n]$ hashes each element in S to a unique location in an array of size n, where each entry of the array stores the fingerprint of the item that hashes to that location. 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. Yet, for many practical applications, perfect hashing is impractical, motivating the search for a different solution.

The first fingerprint hash table that was suggested is called *d-left hashing* [10]. The idea behind d-left hashing is to use a balanced allocation approach. That is, the hash table is partitioned into multiple equally-sized sub-tables, where new elements are placed in the least-loaded sub-table. Balanced allocation allows d-left hash tables to be dimensioned statically so that overflows are unlikely and the average load per block is close to the maximum load.


Index
(1 bit per chain)
Array
chain 2
chain 4
chain 6
A
B
C
D
Empty space
Chain 0 is empty (unset bit)
Chain 6 is not empty (set bit)
B is last in its chain (Set bit)

(a) A block with 8 chains and 4 items. 3 of the chains are non empty (2,4,6) and one of the chains has more than one item (chain 2).


E
A
B
C
D
Chain 0 is now not empty.
Item E is added here, as last of its chain.
All items to the right are shifted 1 place to the right

(b) An example of an addition to the previous block, in this case item E is inserted to the first chain (0). As a result chain 0 is now not empty (bit 0 is set in index). Since chain 0 is the first non empty chain, E is inserted to the first location in the array, shifting other items to the right.

Figure 2: Basic block indexing technique: The first 8 bits are used for index, set bits are marked with light green and unset bits are white. The rest of the bits are allocated to an array that stores remainders with an additional bit that indicates if that remainder is last in chain. Remainders are colored dark green and are marked with letters for easy reference. Let us emphasize that there are *no pointers* in this encoding!

Similarly, cuckoo hashing was also employed in creating efficient Bloom filters [34], with the idea of calculating a perfect hash function using the power of two choices. The main drawback of Cuckoo hashing is that an insertion can cause a large number of memory accesses in order to terminate.

Rank index hashing [25] has an alternative approach. Instead of balancing the load between sub tables, rank index hashing allocates block extensions to overflowing blocks. Statistical multiplexing is used in order to bound the number of required extensions, and the optimal configuration is discovered with an exhaustive search.

TinyTable is another very recent compact hash table based construction [19]. For the approximate set problem (without removals) TinySet has better properties than TinyTable as it offers a better space/accuracy ratio, operates on a fixed sized memory word and dynamically configures its state. Yet, TinySet only supports a limited number of removals and each remove operation slightly degrades its accuracy, whereas TinyTable supports an unlimited amount of removals that do not impact its accuracy. Moreover, TinyTable also has approximate counting and statistics collection capabilities. In contrast, TinySet has no counting capabilities at all.

3 TinySet: Dynamic Fingerprint Resizing

3.1 Motivation and Overview

Our goal is a very space efficient Bloom filter variant that is relatively simple to understand, implement and configure. We would like our data structure to use a single hash function, access a fixed sized memory and degrade performance gracefully as the load increases (like a regular Bloom filter).

We use a similar structure to a blocked Bloom filter. That is, the data structure is partitioned into many fixed size independent blocks. Unlike blocked Bloom filters, however, each of these blocks is not a Bloom filter but a compact representation of a chain based hash table [14]. In a normal chain based hash table, collisions are handled by chaining all items whose hashed values collide. That is, the hash function can be viewed as mapping an item to a given chain, or in other words, returning the index of the chain in which the item is supposed to be located. An item is inserted by adding it to the corresponding chain and it is looked up by scanning the chain pointed to by the hash function.

However, pointers are too expensive in our context. Therefore, we suggest a simple and yet efficient (pointer free) encoding instead, as elaborated below. Further, since the load placed on each specific block fluctuates and cannot be anticipated in advance, we dynamically reconfigure the hash table in order to

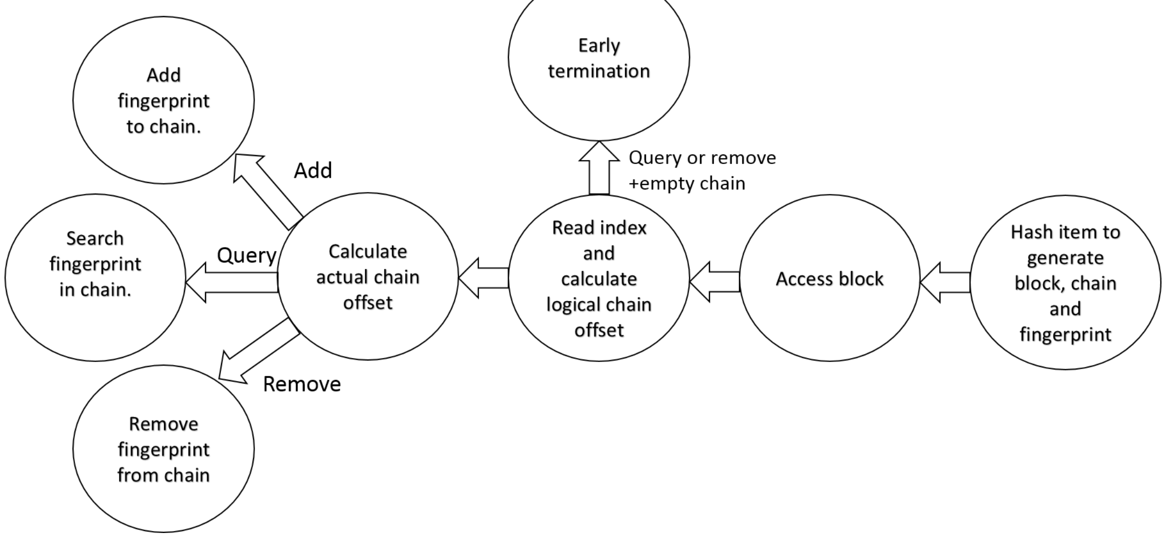

Early termination
Query or remove +empty chain
Add fingerprint to chain.
Add
Search fingerprint in chain.
Query
Calculate actual chain offset
Read index and calculate logical chain offset
Access block
Hash item to generate block, chain and fingerprint
Remove
Remove fingerprint from chain

Figure 3: A flow chart of the basic block operations.

provide attractive false positive rates. That is, each block has a different configuration according to the load placed on it, as illustrated in Figure 1. One of the novelties of this paper is a method to maintain this additional configuration without explicit counters.

3.2 Basic Block Structure

We use a single hash function $H \rightarrow B \times L \times R$, where B is the block number, L is the index of the chain within that block, and R is the *remainder* (or *fingerprint*) to be stored in that block. Unlike traditional hash tables, this value contains pseudo random bits and not the actual key of the item. A block is a continuous, fixed sized memory, and we would like to use it as a compact chain based hash table. We therefore suggest an efficient coding technique.

The first bits in a block contain a fixed size index (I), as illustrated in Figure 2. This index has a single bit per chain in the hash table. If the chain is empty, this bit is unset and vice versa. The rest of the bits in the block are treated as an array(A). This array stores fingerprints extended by an additional indexing bit called the *last bit*, which indicates whether this fingerprint is the last in its chain. Empty chains consume no space in the array. Non empty chains are stored ordered by their chain number.

Our block supports three operations: add, remove and query. The add operation updates the state of the block to include an additional item. Similarly, the remove operation removes an instance of a specific item while the query operation indicates whether or not an item is contained in the block. Similar to Bloom filters, a negative answer is always correct, but a positive one only indicates a fingerprint match and has a false positive probability.

Figure 3 describes a flow chart of these operations. As can be observed, the initial phases are the same for every operation. Specifically, in order to perform an operation on a certain item (T) we first apply the hash function to T. This generates the block number, the chain index and the fingerprint ($B \times L \times R$).

The second step is to access the specific block using the block number. In our design, all the blocks are of fixed size and are continuously aligned in memory. We can therefore simply calculate the block offset and access the block. We then check whether or not the chain we seek is empty. To do so, we access the appropriate bit in the index.

The index is also fixed sized and is placed in the first bits of the block. The bit of the $i'th$ chain is always at offset i. For query and remove operations, encountering an unset bit at this stage allows us to finish the operation as we already know that the item in question is not present. Specifically, a query operation returns false and a remove returns an indication that the desired item was not found.

If we did not terminate early, we first calculate the *Logical Chain Offset (LCO)*. Recall that in the array, non empty chains are stored ordered by chain index. The logical chain offset tells us how many non empty chains are stored before the requested chain. For example, assume that we wish to calculate the LCO of chain 5. Out of chains 0 to 4 only chain 1 and 4 are non empty. Hence, the logical chain offset of 5 is 2 as there are two non empty chains smaller than 5. The offset calculation can be implemented very efficiently with a *rank* operation on the index. Specifically, a $rank(I, c)$ operation returns the number of set bits before the $c'th$ bit. This operation can be implemented efficiently by combining a bit count and a bitwise `and` instructions (both are very efficient). In our example, we calculate $rank(I, 5) = 2$.

Since non empty chains can have more than a single item, our next step is to find the *Actual Chain Offset (ACO)*. ACO is the offset in the array where the chain is stored. To do so, we scan the array and count last in their chain items until we reach the logical offset. At that offset, the desired chain starts (or should start in the case of add to an empty chain).

An initial observation is that the actual offset is always larger or equal to the logical one ($ACO \geq LCO$), as non empty chains by definition have at least a single item (and may have more). Moreover, although the scan operation may seem inefficient, we are interested in blocks that are relatively small and store only a moderate amount of items to begin with. For these parameters, this operation is also cache friendly so we expect good performance (and indeed obtain it as reported in Section 5.2).

Once the ACO is discovered, we can access the required chain. From this point, each operation is different. In particular, a query scans the chain comparing the item's fingerprint to the ones stored in the chain. Add and remove operations are slightly more complicated and require shifting items in order to keep the chains ordered. They are inherently slower, but the overheads are dominated by the block size.

In order to add an item, we first calculate the ACO as detailed above. We then shift all the fingerprints from that offset until the end of the block a single place to the right, and insert the new fingerprint at the ACO offset. Finally, if the appropriate bit in I is unset, we set it and mark the new item as last of its chain.

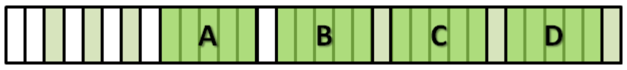
The remove operation is very similar. We first calculate the ACO and then shift all items from the ACO until the end of the block a single place to the left. Finally, if the removed item was marked as last, we either mark the previous item as last in its chain or mark the entire chain as empty. These operations can be implemented in a simple manner. In particular, we first examine the previous last bit. If that bit is set then the removed item was first in its chain. If it is also marked as last we update the index to indicate that the chain is now empty. Similarly, if the previous last bit is unset, the previous item belongs to the same chain as the removed item. If the removed item is marked as last, we can simply mark the previous item as last. No update to the index is necessary here since the chain is still not empty. Finally, if the removed item is not last in its chain, we can simply remove it.

A detailed explanation of the memory layout and indexing technique of a single block is found in Figure 2. Figure 2(a) describes a block with an 8 bit index. Indeed, the first 8 bits are dedicated to the index. Reading the index, we can understand that this block has 3 non empty chains (2, 4 and 6). In the array the chains are always stored sorted by chain number and therefore, the first chain is number 2. For the first item (A), the last bit is unset and therefore A is not the last of its chain. All other items are last in their chains.

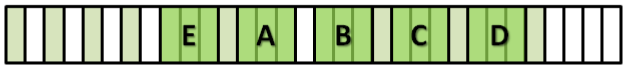
Figure 2(b) illustrates the case where a new item (E) is inserted to chain 0. In this case, since chain 0 becomes the first non empty chain in the block, it is stored at the first location. We shift all the items one place to the right and store E in the first location. Finally, we set E as last of its chain since chain 0 was empty prior to the addition. The remove operation is exactly the opposite. To remove E, we have to shift all items to the right (effectively erasing E), and also clear the index bit of chain 0, basically returning the state to that of Figure 2(a).

3.3 Variable Fingerprint Size

Although the block structure we described is individually very space efficient, it cannot be used efficiently. In particular, a block should be able to accommodate many items since the load fluctuates with some blocks



(a) Before adding E, 4 items in block. We have 24 bits so we allocate 5 bits per fingerprint + 1 additional bit to indicate wether a fingerprint is last in chain.



(b) After adding E, 5 items in block, we now allocate 3 bits per fingerprint + 1 additional bit. Note that not all the space is claimed as we cannot increase the size of all fingerprints.

Figure 4: Dynamic fingerprint size according to load

expected to contain a large number of items. Unfortunately, configuring all the blocks to contain many items results in a lot of wasted space since the majority of blocks are expected to be only near average loaded.

Our approach is to start with a large fingerprint size that has very high accuracy. We then gradually reduce the fingerprint size as the block becomes more crowded. In order to do so, we only need a counter that counts the number of fingerprints in every block. When accessing a block, if there are X fingerprints stored in that block and we know $BlockBitSize$, the number of bits in the block, we can calculate their maximal possible size: $size = \lfloor \frac{BlockBitSize}{X} \rfloor$.

When adding a new item, we need to check whether adding the new item should reduce the size of the fingerprints. Formally, when $\lfloor \frac{BlockBitSize}{X} \rfloor \neq \lfloor \frac{BlockBitSize}{X+1} \rfloor$, a block resize operation is called. It reduces the size of all stored fingerprints in the block in order to make room for an additional fingerprint.

An example of the block resize algorithm appears in Figure 4. In this example, each block has 8 chains and the block is allocated an additional 24 bits. Since the block contains the same items as in the previous example, we only show the memory alignment of items (the logical structure is the same as previously). In Figure 4(a), the block contains only 4 items and therefore each item is allocated 6 bits: 5 bits for fingerprints and 1 bit to indicate if it is last in chain. Figure 4(b) describes the state of the table after an additional item (E) is inserted. Since the number of items is now 5, each item can only be 4 bits long. This means that fingerprints can only be 3 bits long.

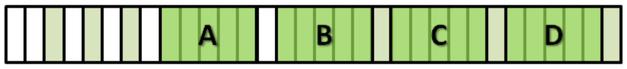
3.4 Two Fingerprint Sizes in One Block

Taking a closer look at Figure 4(b) reveals additional unused bits. We can therefore potentially achieve a slightly better accuracy for the same size. The reason for the unused space is that we use the same size for every item in the block. This is when there are not enough unused bits in order to increase the size of all items by 1.

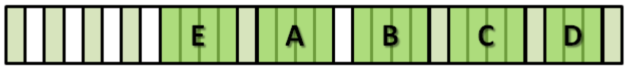
We propose to allow two item sizes per block by calculating how many of the fingerprints can be extended by 1 bit. Specifically, we calculate a second value $mod = X \bmod BlockBitSize$. In workloads that only add items, fingerprints are only shifted to the right and we can always store the first mod fingerprints along with an additional bit.

In order to access the i^{th} item, the calculation changes in the following way: If $i > mod$, we add mod bits, as the first mod fingerprints are 1 bit larger, whereas for $i \leq mod$, we add i bits. Similarly, the size of the extracted fingerprint is $size + 1$ for $i < mod$ and $size$ otherwise.

Figure 5 illustrates the memory alignment of the block before and after the addition of E. Before adding E, we used size=5, mod=0. This means that all the fingerprints are of size 4. However, after we add E, size=3 and mod=4. That is, we are able to store the first 4 fingerprints with size=4. Further, notice that all the bits in the array are now utilized. As long as no removals are present, this is now always the case for our blocks. This optimization makes the resize operations more frequent, as the fingerprints are now resized after any addition. However, these operations are usually less complex since in most cases, only a small number of fingerprints actually change their size.

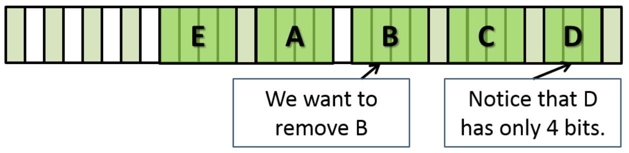


(a) Before adding E, we have 24 bits in the array, and 4 items, therefore all items are of size 5 + last bit

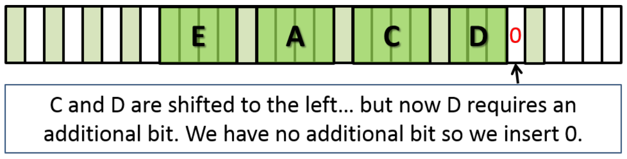


(b) After adding E, we have to split our 24 bits between 5 items. We therefore use, size =3, and mod=4. The first 4 fingerprints are now longer than before.

Figure 5: Adding items with size and mod



(a) Before removing B, in this example, size=3 and mod =4, notice therefore that D is only 3 bits + last bit


E
A
C
D
0
C and D are shifted to the left... but now D requires an additional bit. We have no additional bit so we insert 0.

(b) Removing B, caused us to shift left C and D, however this time D should be mod extended. We cannot make D longer, so we extend it with 0 instead

Figure 6: Removing an item with size and mod

3.5 Removing Items

Unfortunately, since we only store fingerprints of items, we cannot extend them after a removal. However, we can still support removals. Upon a removal, we update the index and shift fingerprints as usual, yet we do not reduce the number of stored items. This way, the size and mod of the existing items are calculated correctly. In order to perform an addition, we first check if there are removed items. If there are such items, we perform an addition without resizing the fingerprints and incrementing the number of items per block. If there are no removed items, we perform the regular add operation, which downsize the fingerprints and increment the number of items.

We now suggest a simple method of checking if there are removed items in the block. It relies on the observation that if there are no removed items, the bitwise array is full, and the last item in the array is always last in its chain. Therefore, the last bit of the array is always set when there are no removals, meaning that testing this bit is a quick indicator to the state of the block. After a removal, since we shift items to the left, the last location in the array is zeroed and the following add operation can reclaim this space.

Also note that when removals are involved, the number of items per block counter monitors the maximal amount of stored items in the block during the operation of the data structure. For brevity, when we discuss the number of stored items in a block we use the term *actual capacity* to describe how many useful items are stored in a block and the term *logical capacity* to describe to how many items the block is configured for. Under constant removals and additions, the actual capacity remains the same but the logical one slowly increases and TinySet becomes less space efficient.

There is an additional delicate point to consider when removals are presents regarding the two sized

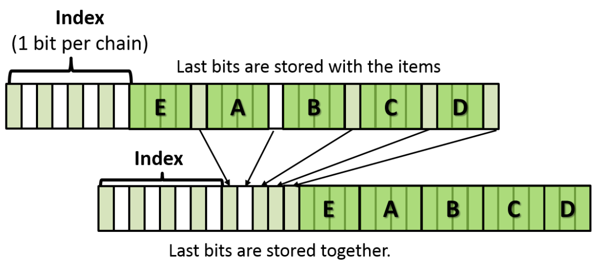

Index
(1 bit per chain)
Last bits are stored with the items
E
A
B
C
D
Index
E
A
B
C
D
Last bits are stored together.

Figure 7: Implicit size counters memory alignment

fingerprints per block optimization. For example, in Figure 6, item B is removed and as a result item D is shifted from a non mod extended location to an extended one. In order to read D correctly, we need to add a 0 bit to D. However, when querying for D, the mod bit may be 1. Therefore, padding D with 0 may result in a false negative that we wish to prevent. Thus, when aligning the items, we only treat the extended bit of a stored item if it is 1. Since naturally only 50% of these bits are 1, supporting remove operations makes the two item sizes per block optimization $\approx$ 50% less efficient. In conclusion, a TinySet that supports removals requires slightly more space for the same accuracy.

3.6 Implicit Size Counters

While conceptually simple, maintaining an explicit items counter is a bit wasteful in situations where the block size is small. We therefore suggest a method to completely eliminate size counters. The idea is to always align the "isLast" bits at the beginning of the array so that we can read them knowing the item size. This idea is illustrated in Figure 7. Notice that the indexing method is the same but the location of the bits is changed.

We can now calculate the number of stored items simply by counting last bits until we reach the last non empty chain. This may seem a bit wasteful at first, but we can also calculate the ACO while doing so. For example, in Figure 7, there are 4 non empty chains. We therefore count bits in the array until we reach the 4th set bit. This happens after 5 bits and therefore there are 5 items in the array. Moreover, while calculating the size, we can also calculate the ACO. In particular, if we wish to access chain 5, we first calculate the logical chain offset. In Figure 7, this offset is 2. That is, after seeing 2 set last bits, we can write down the offset as the ACO. In our example, the the ACO of chain 5 is 3. Hence, in the same operation, we both calculate the number of items in the block and the ACO.

3.7 Efficient Implementation

In theory, the implicit size counters requires us to count all the isLast bits in a block just to discover the number of stored items. This process is linear in the amount of stored items and a straight forward implementation is therefore a bit slow.

However, in practice, for the TinySet configurations discussed in this paper we can significantly speed up this process. We leverage on the following simple observation: If there are k non empty chains in the bucket, then there are at least k items. That is, when counting the isLast bits we can count the first k bits by using a single rank operation. This operation combines a pop count instruction and a bitwise `and` instruction.

In the configurations we suggest, the average chain size λ is smaller than 1, and therefore the vast majority of chains contains just 1 or 0 items in the first place and are counted with a single rank operation. The linear search therefore performs a single step for every item that is not first in its chain, and actually performs just a small number of steps on average.

This idea is explained in Figure 8. We perform two rank operations. The first one counts non empty chain, while the second one multiple isLast bits at once. In the provided example, as there are 5 items in 4

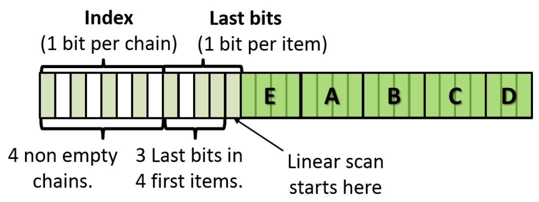


Figure 8: Efficiently counting the number of items on the same example. We perform two rank operations, the first counts the number of non empty chains and the second speeds up the linear scan. In this example, there are 4 non-empty chains and 5 items. That is, the linear scan only performs a single step.

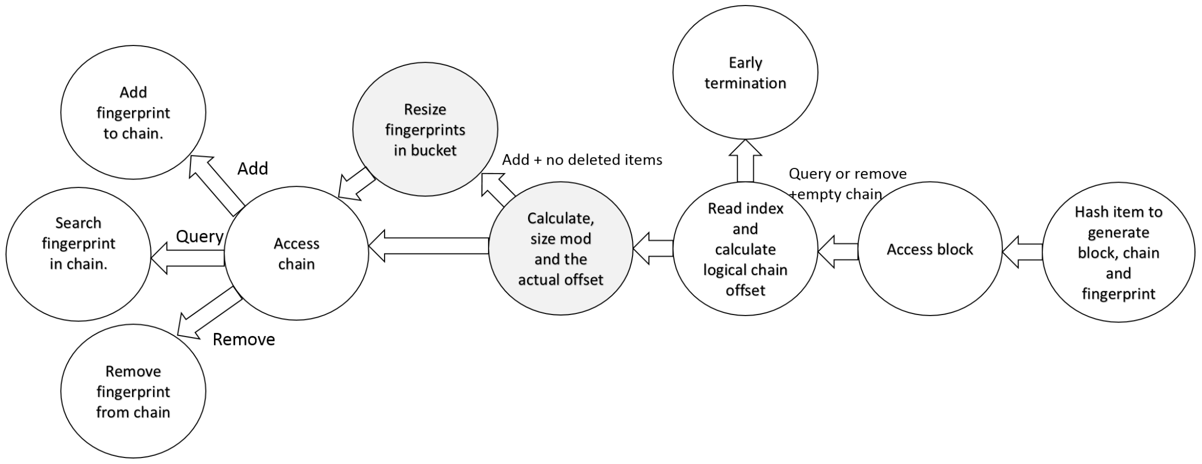

Add fingerprint to chain.
Add
Search fingerprint in chain.
Query
Access chain
Remove
Remove fingerprint from chain
Resize fingerprints in bucket
Add + no deleted items
Calculate, size mod and the actual offset
Early termination
Query or remove +empty chain
Read index and calculate logical chain offset
Access block
Hash item to generate block, chain and fingerprint

Figure 9: A flow chart of TinySet.

chains, the linear scan only performs a single step. When configuring TinySet, we suggest keeping $\lambda < 1$ so that the linear scan remains short.

3.8 Final Overview

We conclude the presentation with a high level overview of TinySet operation. Figure 9 presents the final work flow. The basic operation and concepts are similar to that of the basic block. The main differences are in the gray stages. In particular, before accessing the chain we now calculate the size and mod in addition to the actual chain offset.

Since add operations now downsize fingerprints, we perform a resize operation before executing an add operation. If there are removed items in the block, we simply reclaim a previously removed item and do not perform the resize.

4 Analysis

4.1 Memory Overheads

For a chain based fingerprint hash table with an average chain length of λ and a (fixed) fingerprint size of $Log(\varepsilon)$ bits, the false positive ratio is $\lambda \cdot \varepsilon$ [25]. Therefore, if we pick $\lambda = 1$, then each chain is on average 1 item long and the expected false positive rate is ε. This false positive rate is optimal up to the indexing overheads of the block [25].

In TinySet, we also require a single bit per item and an additional bit per chain. When $\lambda = 1$, our indexing cost is exactly 2 bits per item. A single TinySet block is therefore optimal up to an additive factor of 2 bits per item. Similarly, if the load of all blocks is perfectly balanced, all individual blocks are configured the same and TinySet is only 2 bit per item from being optimal.

Each TinySet block contains a fixed size index (typically 64 or 32 bits) and a fixed size array. Additionally, TinySet blocks may contain an items counter that has to be sized so that overflows are unlikely. This counter can be eliminated with the implicit counters optimization. Hence, the per item bit cost of TinySet is simply $\frac{BlockSize+CounterSize}{itemsPerBlock}$.

4.2 Variable Sized Fingerprints

In order to calculate the accuracy of TinySet, we average the accuracy for each block. We start by calculating the false positive rate of a single block with r fingerprints. Consider $\lambda_r = \frac{r}{L}$ the local λ of a block with r fingerprints. We first analyze the more simple case where all fingerprints in a single block are of the same size.

For each possible fingerprint size $S_1, .., S_k$, we calculate the maximal number of fingerprints that can be stored with this size, $C_1, ..., C_k$, subject to that block bit size. That is, if there are $BlockSize$ bits allocated for the array, $C_i = \left\lfloor \frac{BlockSize}{S_i+1} \right\rfloor$.

Denote p_r the probability that there are exactly r fingerprints in a block. Assuming we already know p_r, we can calculate the contribution of each size to the overall error. This impact is: $Error(size = S_i) = \sum_{r=C_{i-1}+1}^{r=C_i} p_r \cdot \frac{\lambda_r}{2^{S_i}}$.

In order to calculate p_r, we consider the classic balls and bins experiment, where n balls are thrown randomly into m bins. The probability that a certain bin contains exactly r balls is:

$$p_r = \left(\begin{array}{c} m \\ k \end{array} \right) \left(\frac{1}{n}\right)^r \left(1 - \frac{1}{n}\right)^{n-r} = \frac{1}{r!} \frac{m(m-1)...(m-r+1)}{n^r} \left(1 - \frac{1}{n}\right)^{m-r} \approx \frac{e^{-m/n}(m/n)^r}{r!}$$

We therefore conclude that the false positive rate is $FP = \sum Error(size = S_i)$.

4.3 Variable Sized Fingerprint With Mod

Next, we analyze the case of two item sizes in the same block. Denote m_r the probability of a fingerprint to be modulo expanded when r fingerprints are stored in the block ($m_r = \frac{mod_r}{r}$). We sum up the contribution to the error of each possible size in a similar way as before: $Error(size = S_i) = \sum_{r=C_{i-1}+1}^{r=C_i} p_r \cdot (1 - m_r) \cdot \frac{\lambda_r}{2^{S_i}} + \sum_{r=C_{i-2}+1}^{r=C_i-1} p_r \cdot (m_r) \cdot \frac{\lambda_r}{2^{S_i}}$. The false positive rate remains $FP = \sum Error(size = S_i)$.

4.4 Overflows

TinySet has no significant overflow problem, since we can size the items counter reasonably. Yet, overflow is unavoidable in very extreme cases, where so many fingerprints are inserted to a block that there is no more room to allocate even a single bit per item. We denote the maximal number of fingerprints a block can contain by Z_{max}. In this extreme cases there are more items than bits in the block. We now analyze this case: Denote X_i, $0 \leq i \leq B$, the number of items inserted to the i'th block, and denote by n the average number of fingerprints a block stores. Denote O the overflow indicator variable: $O = 1$ if there is an overflow, and $O = 0$ otherwise.

$$\Pr\left[O = 1\right] = \Pr\left[(\max X_i) > Z_{max}\right] \leq \sum_{i=1}^{B} \Pr\left[X_i > Z_{\max}\right] = B \times Binotail\left(n, \frac{1}{B}, W_3\right)$$

where $Binotail(N, P, K)$ denotes the tail probability $\Pr\left[Y > k\right]$ and Y has the distribution $Binomial(N, P)$. To get a sense for this, in the above example, the probability of a single block overflow is smaller than 10^{-200}. Therefore, even in extremely big TinySets with billions of fingerprints, this probability is negligible. When size counters are used, we need to intelligently pick their size so that the overflow probability is arbitrarily small and overheads are low. To do so, we perform the same calculation. In particular, if we pick the items

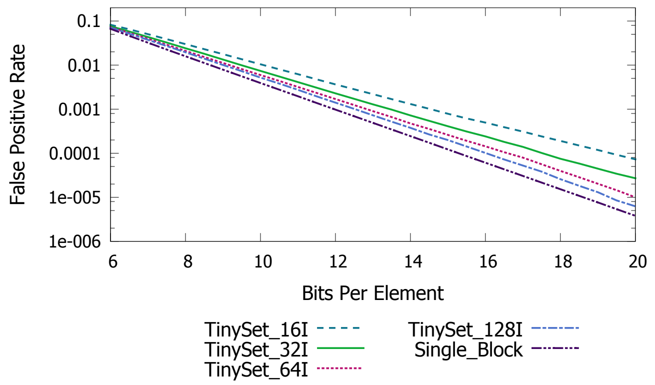


Figure 10: Space/Accuracy tradeoff for different average number of items per block and $\lambda = 1$

counter to be 7 or 8 bits, the overflow probability of a single block with average load of 64 items is less then 10^{-13} and 10^{-23} respectively. The overhead for 8 bits counters is only 0.125 bits per item in the 64 items per block case. Even in this case, it is a significant improvement over previous works that required a different configuration for each overflow probability.

5 Results

We start by comparing our work to rank indexed hashing, Bloom filters and d-left hashing. TinySet requires the following parameters for configuration: the block size, the number of chains per block (L), and the desired average block load(λ) that determines how many blocks to create. Our measurements are calculated with the implicit size counters optimization. That is, if the optimization is not to be used, counter overheads should also be accounted for. Bloom filters and rank indexed hashing both require to know the expected number of elements while rank indexed hashing requires additional parameters. Whenever we present rank indexed hashing or Bloom filters, we configure them optimally for the corresponding data point. When we present d-left hashing, we configure it using the configuration suggested by its authors (8 fingerprints, 4 sub-tables, expected load of 6). We also compare our work to blocked Bloom filters and their improvement balanced Bloom filters; these constructions are configured according to their respective authors instructions.

5.1 Required Block Size

Figure 10 describes the space/accuracy tradeoff of TinySet that is configured to contain a fixed amount of items per block with $\lambda = 1$. That is, as the false positive rate decreases, TinySet blocks become larger as they store on average the same number of (longer) fingerprints.

Table 1 summarizes the required space for different practical false positive ratios for the various configurations. As can be observed, in practice the 64 item configuration consumes only 0.5-1.3 more bits per item than a single TinySet block of the same accuracy, making it a very attractive configuration. Since larger TinySet blocks result in more complex operations, we continue with a focus on block sizes that are smaller than 128. Very small blocks experience a high variance and therefore offer a significantly worse space/accuracy tradeoff. This is anticipated by our theoretical analysis and is a known issue also for blocked Bloom filters.

False Positive	TinySet 16I (L=16, $\lambda = 1$)	TinySet 32I (L=32, $\lambda = 1$)	TinySet 64I (L=64, $\lambda = 1$)	TinySet 128I (L=128, $\lambda = 1$)	Single Block ($\lambda = 1$)
1%	10.1	9.5	9.1	9	8.4
0.1%	15.1	13.5	12.8	12.5	12
0.01%	19.5	17.5	16.6	16.2	15.3

Table 1: Different configurations with a constant average number of items per block compared to the single block case

	Bloom Filter	TinySet_32I	TinySet_64I
1%	0.65/0.69	0.28/0.05	0.34/0.08
0.1%	0.96/0.98	0.28/0.05	0.34/0.08
0.01%	1.33/1.29	0.28/0.05	0.35/0.08

Table 2: Time to add/query 1 million item (seconds)

5.2 Operation Speed

In this section, we study the performance of our Java based implementation of TinySet [6] compared to an open source Bloom filter implementation[1]. Our experiment goes as follow: We measure the time it takes to add 1 million items to each of the constructions. We then perform a single query for each contained item, repeat this 10 times and average the results. We compare the Bloom filter to TinySet 32I and TinySet 64I, the former has 32 items on average per block and the latter 64. The experiment was run on an Intel i7 working at 3.2GHZ. The computer also has 32GB RAM so all the data structures easily fit in main memory.

Our results are in Table 2. As can be observed, both TinySet configurations are faster than a Bloom filter for the false positive range. We also note that the difference is more dramatic for low false positive rates. As anticipated, since the complexity of TinySet operations depends on the block size, TinySet 32I is significantly faster than TinSet 64I.

Figure 11 provides additional understanding about the dynamics of the performance for 0.1% false positive range. In these measurements, we averaged 10 runs of each algorithm for a 1 million items benchmark. As can be observed in Figure 11(a), the add operation is initially orders of magnitude faster than the Bloom filter as the TinySets are empty and the block management overheads are low. As TinySet becomes more crowded, the add operation is slower, until the blocks are nearly full and both configurations are slightly slower than the Bloom filter. For query operations, however, the increase in run speed is more moderate, and both configurations are significantly faster than the Bloom filter, as seen in Figure 11(b).

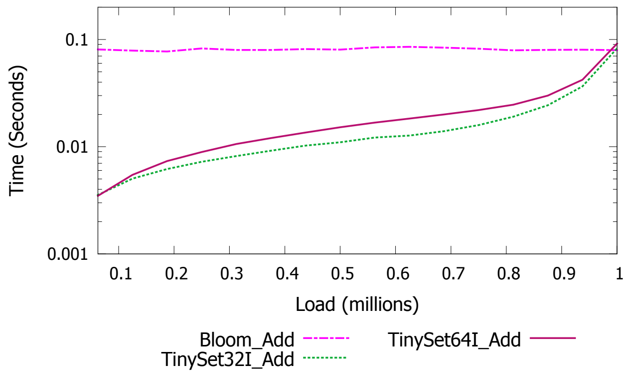
5.3 Space/Accuracy Tradeoff

Table 3 compares TinySet (64 items per block, $\lambda = 1$) to both Bloom filters and rank indexed hashing. The Bloom filter is optimally configured and rank indexed hashing is configured according to the configuration suggested by its authors. For perspective, we also added to the table the lower bound for this problem [12]. As can be observed, this configuration of TinySet is more compact than both Bloom filters and rank indexed hashing.

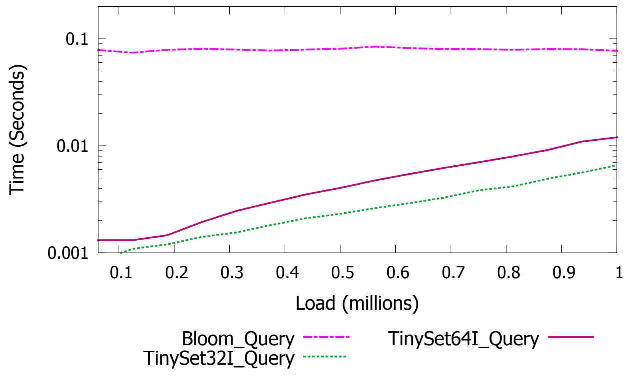
In Table 4, we compare TinySet against a variable increment counting Bloom filter (VI-CBF) [41], d-left and rank indexed hashing that supports removals. We note that this is not an entirely fair comparison, as removals gradually make TinySet less space efficient. To complete the picture, Section 5.6 quantifies what happens to TinySet's efficiency when removing items. As can be observed, TinySet is initially $18-27\%$ more space efficient than the best alternative for the range. Yet, as this advantage degrades over time we conclude that it may be attractive as a CBF only for workloads with limited number of removals.

In Figure 12, we configured TinySet to contain an average of 40 items per block ($\lambda = 0.625$). Fixing this parameter, we analyzed the impact of the number of bits allocated per item on the accuracy using both our

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



(a) Add Operation, 0.1% false positive



(b) Query Operation,0.1% false positive

Figure 11: Time to perform 62.k operations as a function of the load, at 1 million items TinySet 64I has 64 items on average per block and similarly TinySet 32I has on average 32 items per block.

analysis and a simulation exercising the real code. The experiment was repeated 100 times and averages are reported. As can be seen, the analysis is accurate and TinySet is superior to both rank indexed hashing and Bloom filters. For Bloom filters, the break even point is ∼7.9 bits per item; from this point on, TinySet offers a better tradeoff than Bloom filters. To quantify this, we note that for 19 bits per element, this TinySet configuration is over 4 times more accurate than the best alternative.

5.4 Memory Access Efficiency

TinySet always uses a bounded amount of space for all its operations, a property that has benefits in both hardware and software implementations. We therefore continue our evaluation with configurations for a fixed block size. We note that Bloom filters, rank indexed hashing and d-left hashing cannot guarantee that a read/update operation are bounded to a single memory word.

An attractive configuration for a software implementation is to size the blocks to the same size as a cache line. In that case, a meticulous implementation can guarantee that the blocks are perfectly aligned to cache lines. Since 64 bytes is the cache line size of Intel's 64 and IA-32 architecture [3], we suggest to configure TinySet blocks to this size. We call this configuration TinySet 64B.

We compare this TinySet configuration with other access efficient alternatives, namely BlockedBF and BalancedBF. Recall that BalancedBF also employs a load balancing scheme to improve the space efficiency. For BalancedBF the configuration we used is called *single* by the authors of [29] as it is limited to a single memory access (in addition to an access to the overflow list). In this configuration, the size of the overflow

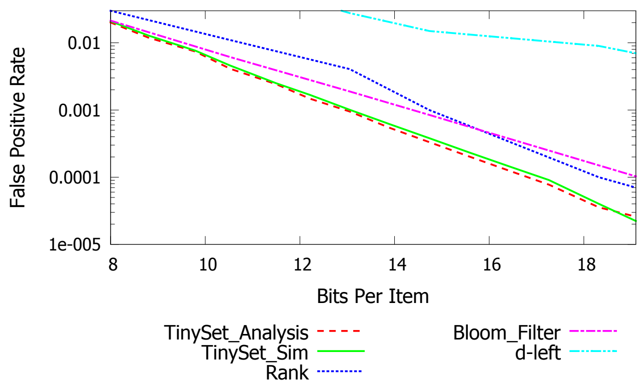


Figure 12: Space/Accuracy Tradeoff for different algorithms.

False Positive	TinySet 64I (L=64,$\lambda = 1$)	Rank	Bloom Filter	Lower Bound (Information Theory)	Comparison vs Rank	Comparison vs Bloom Filter	Comparison vs Lower Bound
1%	9.1	10.6	9.6	6.4	-14%	-5%	+38%
0.1%	12.8	14.4	14.4	10	-11%	-11%	+28%
0.01%	16.6	18.2	19.1	13.3	-9%	-13%	+25%

Table 3: Required storage (in bits) per element for the same false positive rate (without removals)

list is 0.49% of the items. Since both BlockedBF and BalancedBF are evaluated with 32 bytes memory words, for a fair comparison we also created a 32 byte configuration for TinySet (TinySet 32B).

Table 5 presents our 64 bytes TinySet configurations and compares the results to a standard Bloom filter. As can be observed, this configuration is more space efficient than a Bloom filter for the range. We note that both BlockedBF and BalancedBF can never be as space efficient as a standard Bloom filter.

Also, note that the three bottom configurations in Table 5 are actually the same configuration under different load settings. Although slightly less space efficient for 1% false positive ratio, TinySet can use the same configuration for the entire false positive range. We further describe this capability in Section 5.5.

Table 6 describes a single TinySet configuration that is very efficient for 32 byte memory blocks. We compare it to a BalancedBF. As can be observed, this configuration requires $9 - 18\%$ less space for the same false positive rate compared to balanced Bloom filters. Figure 13 gives a wider perspective on the comparison between TinySet and the alternatives. As can be observed, both TinySet configurations yield significantly lower false positive for the same space. At the end of the range, the error of TinySet 32B is ≈ 9 times more accurate than the best alternative while TinySet 64B is over 30 times more accurate (note that it uses a larger memory word).

We conclude that TinySet is also more space efficient than access efficient Bloom filters while also providing access efficiency guarantee. It is also important to note that load balancing techniques similar to the one suggested by BalancedBF can also further improve the space/accuracy tradeoff of TinySet.

5.5 Flexibility

Unlike other hash table constructions, TinySet is very flexible and as the load increases, TinySet behaves in a similar way to a Bloom filter. Further, similar to a Bloom filter, TinySet can easily accommodate more items than anticipated and its accuracy degrades gracefully. In such cases, other hash table solutions typically overflow, and therefore cannot operate correctly.

Figure 14 describes an experiment where we configure both Bloom filter and rank indexed hashing optimally for a specific amount of items (38K items). Rank indexed hashing is configured with 1k blocks, and similarly TinySet is configured with 1K blocks. Overall, all constructions are allocated exactly the same

False Positive	TinySet 64I (L=64,$\lambda = 1$)	Rank CBF	d-left CBF	VI-CBF	Comparison vs Rank CBF	Comparison vs d-left CBF	Comparison vs VI-CBF
1%	9.5	13	17.6	25	-27%	-46%	-62%
0.1%	13.2	16.8	22.3	37.8	-21%	-41%	-65%
0.01%	16.9	20.6	26.4	50	-18%	-36%	-66%

Table 4: Required storage (in bits) per element for the same false positive rate (with removals)

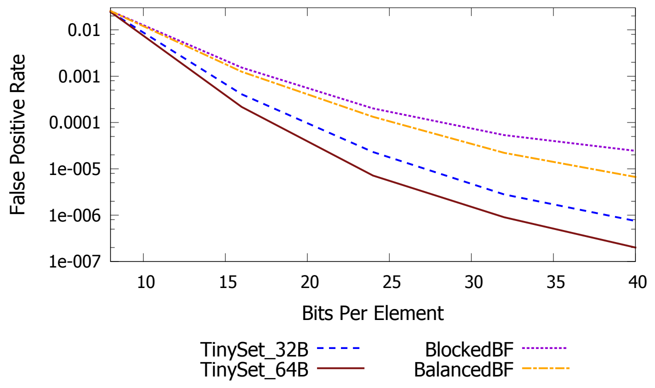


Figure 13: TinySet with fixed block size compared to access efficient Bloom filters.

amount of space. We insert items to the data structures and evaluate their accuracy as the load increases. We continue doing so even after the anticipated load is achieved.

As can be observed, TinySet offers better space accuracy tradeoff than both rank indexed hashing and the Bloom filter. While at the anticipated load, all constructions offer a very similar space/accuracy tradeoff, rank indexed hashing is less accurate throughout the experiment and cannot continue without overflowing as the load increases. TinySet remains more accurate than the Bloom filter until the end of the experiment, with a false positive rate of 3.3%. If we configure a Bloom filter optimally to the eventual load, we should use two fewer hash functions. In this case, the Bloom filter can be configured to be slightly more accurate than TinySet.

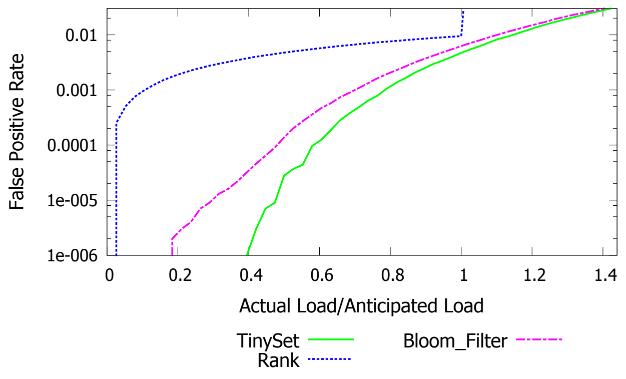


Figure 14: Performance under increasing load.

Configuration Details				Space (bits per item)		Comparison
False Positive	L	λ	Array Size	TinySet 64B	Bloom Filter (not access efficient)	vs Bloom Filter
1%	80	0.7	432 bit	9.1	9.6	-5%
	64	0.87	448 bit	9.2		-4%
0.1%		0.61		13.1	14.4	-9%
0.01%		0.45		17.7	19.1	-7%

Table 5: Attractive TinySet configurations for 64 bytes per block compared to a standard Bloom filter

Configuration Details				Space (bits per item)			Comparison	
False Positive	L	λ	Array Size	TinySet 32B	BalancedBF (a=1,$\gamma = 0.0049$)	BlockedBf	vs BalancedBF	vs BlockedBF
1%	32	0.84	224	9.5	10.4	10.9	-9%	-17%
0.1%		0.57		14	17	17.4	-18%	-20%
0.01%		0.41		19.6	24	27.5	-18%	-29%

Table 6: Attractive TinySet configurations for 32 bytes per block compared to access efficient Bloom filters

5.6 Removals

As we stated above, frequent removals degrade the space efficiency as we cannot increase the size of fingerprints after removal. However, TinySet can reasonably support a moderate number of removals and still provide competitive accuracy. In the following experiments we test the total amount of items stored in TinySet compared to the logical amount that remains the same. That is, once TinySet is full, at each step we add an item and remove an item. We tested two removal patterns, a sliding window where the oldest entry is removed and a random removal pattern.

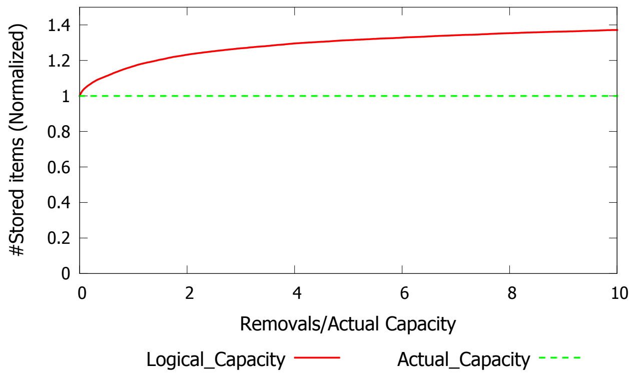
Figure 15 illustrates the results of these measurements. As can be observed, the behavior of TinySet under these two workloads is almost identical. At the beginning, removed items are infrequent and therefore we usually add new items and indeed we see a sharp increase in the number of stored items. In particular, after 50% of the items are replaced, TinySet contains $\approx 11\%$ removed items. However, over time removed items become more frequent and after 100% removals 16% of the items stored are removed items. Taken to the extreme, after 1000% of the items are replaced (the number of remove operations is 10 times the number of stored items), $\approx 35\%$ of the items are removed items. Since TinySet is a very space efficient to begin with, it can remain competitive for a while and may also be attractive as a counting Bloom filter for some applications.

6 Conclusions and Discussion

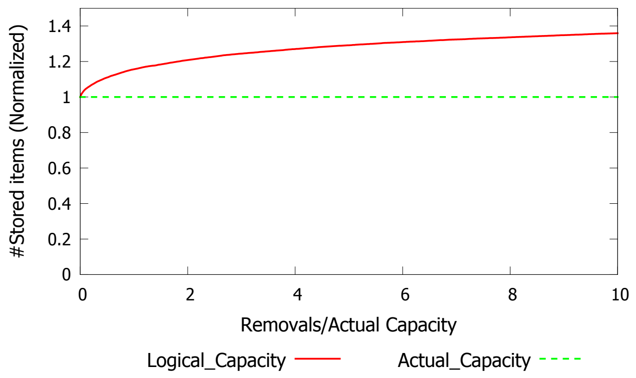
In this work, we have introduced TinySet, an alternative Bloom filter construction that combines several appealing properties, namely access efficiency, speed, space efficiency and partial support for removals. To the best of our knowledge, it is the only mechanism that provides all of these at once. Interestingly, TinySet is both faster and more space efficient than plain Bloom filters, especially for query operations. We also demonstrated that TinySet is more space efficient than both access efficient Bloom filters and other hash table based counting Bloom filters suggestions.

TinySet's access efficiency comes from the fact that its operations only access a single fixed sized block, which can be configured to match a single cache line. We achieve this by employing a novel indexing technique that dynamically downsizes the stored fingerprints as the load increases. Since the load fluctuates between the blocks, each one has a slightly different local configuration. When no removals are present, each block is 100% utilized in order to provide the best possible accuracy.

While removals gradually degrade TineySet's space efficiency (since we have no way of making fingerprints longer once the load decreases), we have showed that in practice, TinySet remains relatively space efficient



(a) Sliding Window



(b) Random Removal

Figure 15: Effect of removals on TinySet memory space utilization.

for a large number of remove operations under different removal patterns.

Finally, we showed that TinySet is more flexible than a Bloom filter, as it can start with very long fingerprints and gradually downsize them as the load increases. Bloom filters, on the other hand, cannot dynamically change the number of hash functions they use. Notice that TinySet's flexibility also prevents it from overflowing, unlike other hash table based constructions.

In the future, we would like to extend TinySet's functionality and combine it with counter compression methods such as [16]. A Java based open source implementation of TinySet and all other code used in this paper is available at [6].

References

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

[2] https://bitcoinfoundation.org/blog/?p=16.

[3] Intel 64 and ia-32 architectures optimization reference manual. http://www.intel.com/content/dam/www/public /us/en/documents/manuals/ 64-ia-32-architectures-optimization-manual.pdf.

[4] Mellanox ib qdr 324p switch system - overview.

[5] Squid. http://www.squid-cache.org/.

[6] TinySet implementation. https://code.google.com/p/tinyset/.

[7] N. S. Artan, K. Sinkar, J. Patel, and H. J. Chao. Aggregated bloom filters for intrusion detection and prevention hardware. In *IEEE GLOBECOM 2007*.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 1970*.

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

[10] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms 2006*.

[11] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, 2002.

[12] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman. Exact and approximate membership testers. In *ACM STOC 1978*.

[13] 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. 2008*.

[14] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press.

[15] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. ACM SIGCOMM 2003.

[16] G. Einziger, B. Fellman, and Y. Kassner. Independent counter estimation buckets. In *The 34th Annual IEEE International Conference on Computer Communications (INFOCOM 2015)*, Hong Kong, P.R. China, Apr. 2015.

[17] G. Einziger and R. Friedman. Postman: An elastic highly resilient publish/subscribe framework for self sustained service independent p2p networks. In *Springer SSS 2014*.

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

[19] G. Einziger and R. Friedman. Counting with TinyTable: Every bit counts! Technical report, Computer Science Department, Technion, 2015.

[20] G. Einziger, R. Friedman, and Y. Kantor. Shades: Expediting kademlia's lookup process. In *International Conference on Parallel Processing (Euro-Par)*. 2014.

[21] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. on Netw.*, 2000.

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

[23] S. Geravand and M. Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks 2013*.

[24] M. T. Goodrich and M. Mitzenmacher. Invertible bloom lookup tables. *CoRR*, abs/1101.2245, 2011.

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

[26] K. Huang, J. Zhang, D. Zhang, G. Xie, K. Salamatian, A. Liu, and W. Li. A multi-partitioning approach to building fast and accurate counting bloom filters. In *IEEE IPDPS 2O13*.

[27] Z. Jerzak and C. Fetzer. Bloom filter based routing for content-based publish/subscribe. In *Proc. of the 2nd International Conference on Distributed Event-based Systems*, DEBS, pages 71–81. ACM, 2008.

[28] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander. Lipsin: Line speed publish/subscribe inter-networking. *ACM SIGCOMM 2009*.

[29] Y. Kanizo, D. Hay, and I. Keslassy. Access-efficient balanced bloom filters. *Computer Communications*, 36(4):373 – 385, 2013.

[30] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices 2000*.

[31] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS 2010*.

[32] T. Lee, K. Kim, and H.-J. Kim. Join processing using bloom filter in mapreduce. In *ACM RACS 2012*.

[33] L. Li, B. Wang, and J. Lan. A variable length counting bloom filter. In *ICCET 2010*.

[34] S. Lumetta and M. Mitzenmacher. Using the power of two choices to improve bloom filters. *Internet Mathematics*, 4(1):17–33, 2007.

[35] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB 1983*.

[36] M. Mitzenmacher. Compressed bloom filters. In *ACM PODC 2001*.

[37] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 2008.

[38] A. Pagh, R. Pagh, and S. S. Rao. An optimal bloom filter replacement. In *ACM-SIAM SODA 2005*.

[39] F. Putze, P. Sanders, and J. Singler. Cache-, hash- and space-efficient bloom filters. In *WEA*, pages 108–121, 2007.

[40] Y. Qiao, T. Li, and S. Chen. One memory access bloom filters and their generalization. In *IEEE INFOCOM 2011*.

[41] O. Rottenstreich, Y. Kanizo, and I. Keslassy. The variable-increment counting bloom filter. In *IEEE INFOCOM 2012*.

[42] O. Rottenstreich and I. Keslassy. The bloom paradox: When not to use a bloom filter? In *IEEE INFOCOM 2012*.

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