Stretching Your Data With Taffy Filters

Jim Apple

Abstract

Popular approximate membership query structures such as Bloom filters and cuckoo filters are widely used in databases, security, and networking. These structures represent sets approximately, and support at least two operations – insert and lookup; lookup always returns true on elements inserted into the structure; it also returns true with some probability $0 < \varepsilon < 1$ on elements *not* inserted into the structure. These latter elements are called false positives. Compensatory for these false positives, filters can be much smaller than hash tables that represent the same set. However, unlike hash tables, cuckoo filters and Bloom filters must be initialized with the intended number of inserts to be performed, and cannot grow larger – inserts beyond this number fail or significantly increase the false positive probability. This paper presents designs and implementations of filters than can grow without inserts failing and without meaningfully increasing the false positive probability, even if the filters are created with a small initial size. The resulting code is available on GitHub under a permissive open source license.

1 Introduction

The Bloom filter is a ubiquitous data structure that allows storing a set with a low amount of space. Bloom filters support the operations insert – which adds an item to the set – and lookup, which returns true if an element is in the filter; if an element is not in the filter, true is returned with some configurable probability $0 < \varepsilon < 1$. This is called the "false positive probability", or "fpp".

There are a number of other structures also supporting insert and lookup with a false positive probability greater than 0 [3, 4, 8, 22, 33, 50, 59]. A lookup operation with these guarantees is sometimes called an "approximate membership query", and structures that support approximate membership queries are sometimes referred to "AMQ structures" or just "filters". The significant interest in filters is reflective of their utility in applications such as databases, security, and networking [1, 4, 9, 11, 18, 27, 31, 32, 59].

Each of the filter structures cited above supports approximate membership queries on sets with a given maximum size, but the question of extensible (or *extendable* or *incremental* or *growable*) filters that can increase in capacity as more elements are inserted is little studied. The classic answer is to create a sequence of filters, possibly of increasing sizes and/or lower false positive probabilities [2, 13, 34]. Inserts occur on the last filter to be created and lookups must search each filter. Even in designs for which this keeps the false positive rate low, lookup times balloon from constant to poly-logarithmic or even linear in n, the number of elements inserted [45, 48, 63]. Additionally, the space usage often grows as $\Omega(n \lg n)$, at which point a traditional hash table would do the same work in the same space with constant-time operations and an n^{-c} false positive probability, where c depends on the constant in $\Omega(n \lg n)$. A newer approach to manage growing filters is to use cuckoo or quotient filters in which, each time the filter capacity grows, the false positive probability doubles [8, 49, 59, 62, 63]. Finally, a third approach to the problem of growing a filter is to depend on the original keys being available during rebuild time [60]. This approach is not always possible or time efficient. See Figure 1, which describes prior work and its limitations when filters grow.

Instead of these approaches, this paper investigates practical structures that allow the structure to grow and keep a low false positive rate (not exceeding a threshold specified when the structure was created), all while using no more than $O(\lg \lg n + \lg(1/\varepsilon))$ bits of space per element [48]. This is a significant improvement over the status quo in which filters either cannot grow, such as standard Bloom filters, or use $\lg(1/\varepsilon) + \omega(\lg \lg n)$ or $\omega(\lg(1/\varepsilon))$ bits to represent sets with size n and false positive probability ε.

1.1 Applications

Growable filters are potentially useful in situations where there is no known bound on the number of keys to be inserted. One example is in joins in query processing systems. It is often beneficial to performance to create and populate a filter

Behavior	Filters
$\omega(1)$ lookup	dynamic bloom [34], scalable bloom [2], dynamic cuckoo [13], monkey [16]
Doubled fpp when capacity doubles	logarithmic dynamic cuckoo [63], Morton [8], vacuum [59], rank select quotient [49], consistent cuckoo [45], dynamic cuckoo [13], entry-extensible cuckoo [62]
Depend on storing $\Omega(\lg n)$ bits per element (in filter or in backing store)	elastic [60], consistent cuckoo [45], Chucky [17]
More than double fpp when full beyond capacity	Bloom [5], tinySet [23]

Figure 1: The filter types that exhibit various undesirable behavior as more keys are inserted

for the build side of a join: the filter, being much smaller than the full output of the build-side hash table construction, can be pushed-down to the probe side to reduce the number of rows that need to be tested against the build output [7]. If there are any predicates on the build side, or if the build side has incomplete or inaccurate distinct value count statistics, it is not possible to predict the eventual size of the filter. Systems like Apache Impala estimate the cardinality when initializing the filter and then discard the filter if the estimate was too low [40]. Using growable filters would allow these filters to continue to be populated and used in the probe side.

Another example where growable filters are useful is in log-structured merge trees ("LSM trees") [44]. Log-structured merge trees store data in sorted "runs" of exponentially-increasing size. In order to cheaply discover if a key is present in a run, systems like RocksDB equip each run with a filter [22, 44]. Point lookups that go through the filters require accessing $\lg n$ filters, where n is the number of keys in the LSM tree. A single growable filter structure can reduce this to a single filter query by storing one structure for all keys, rather than $\lg n$ structures. Here a Bloomier filter (sometimes called a retrieval data structure) is called for, in which every positive result from a lookup operation has an attached value [12]. For LSM trees, that value should be the identifier of the most-recently-created run a key is associated with. Upon a positive lookup in the filter, the run identifier is retrieved, and a more expensive probe of that run can begin.[1]

A final example is previously-used passwords [57]. The goal of a filter for these cases is to allow lookups during password creation time and prevent users from using a previously used password. These sets can have long lifetimes and grow arbitrarily large; the "Have I Been Pwned" data set is 11GB of SHA-1-hashed passwords [35]. Because of password databases' propensity for growth, static-capacity structures like Bloom filters or cuckoo filters are less well suited for these data sets. Section 6.3 discusses this example in more detail.

---

[1]The "Chucky" system is built on this premise, but requires a full filter rewrite at each last-level compaction [17]. SlimDB also uses a cuckoo filter to implement a retrieval structure on the most-recently-created "sub-level" that a key is in in an LSM, but doesn't use dynamic sizing at all [55].

1.2 Contributions

To address the need for filters that can grow, this paper makes three contributions.

1. Section 3 presents the *taffy block filter* ("TBF"), a Bloom-filter-backed AMQ structure with $O(\lg n)$ lookup cost.

2. Section 4 presents the *taffy cuckoo filter* ("TCF"), a cuckoo-hashing-based AMQ structure with $O(1)$ lookup cost.

3. Section 5 presents the *minimal taffy cuckoo filter* ("MTCF"), a cuckoo-hashing-based AMQ structure that decreases the space needed in a TCF by up to a factor of 2.

TCFs, in addition to having $O(1)$ lookup, contribute a new understanding of cuckoo filters as dictionaries. MTCFs apply for the first time the technique of quotienting to dictionaries that can grow without doubling in size, which may be of independent interest.

Section 6 describes experimental performance results on all three taffy filters and what circumstances each is suited for. Section 7 concludes.

2 Prior work

2.1 Split block Bloom filters

The insert and lookup operations in standard Bloom filters access $\lg(1/\varepsilon)$ bits in an array of size m that stores $m \ln 2/\lg(1/\varepsilon)$ distinct elements [6]. These cause $\lg(1/\varepsilon)$ memory accesses and require the same number of hash function applications. Block Bloom filters reduce the number of memory accesses to 1 at a cost of a slightly increased false positive probability [53].

Each block Bloom filter is implemented as an array of non-overlapping blocks; see Figure 2. Each block is itself a Bloom filter. Blocks are no larger than a single cache line in size. To insert a key, the key is hashed to select the block to use, mapping a key x to $h(x) \bmod m/B$, where h is the hash function, m is the size of the block Bloom filter and B is the size of each block.

Symbol	Usage
a	The logarithm, base 2, of the number of buckets in an array in a TCF or an MTCF.
b	The number of slots in a bucket in a filter or hash table that uses buckets.
B	The size of a block in a block Bloom filter.
d	The over-provisioning per key - the number of bits per element that need to be stored beyond $\lg(1/\varepsilon)$.
F	The size of fingerprints in TCFs and the size of large fingerprints in MTCFs. See Sections 4 and 5.
k	The number of hash functions in a cuckoo hash table or Bloom filter.
L	The size of a "lane" in a split Bloom filter.
m	The number of bits in a Bloom filter.
n	The number of keys in a filter or dictionary at a given point in time.
N	The maximum number of keys that will ever be in a filter. Always less than $|U|$.
p	The logarithm, base 2, of the number of levels in an MTCF.
S_i	The set of permutations on the integers in $[0, i)$.
T	The maximum size of tails in TCFs and MTCFs. See Sections 4 and 5.
U	The "universe" - the set of keys that could be put in a filter.
$\mathbb{Z}_i$	The set of integers $[0, i)$.
$\mathbb{Z}_2^i$	The set of bit strings of length i.
δ	The over-provisioning per structure - the percent of empty space in a dictionary or filter.
ε	The false positive probability, or "fpp".
φ_i	The permutations associated with side i of a TCF. See Section 4.
$A \uplus B$	The tagged union of A and B such that even if $A \subseteq B$, $A \uplus B \neq B$.

In split block Bloom filters, once a block is selected, it is used as a "split" Bloom filter [9]. In a standard Bloom filter, to insert a key x, $k = m \ln 2/n$ hash functions are applied to x, and each bit $h_i(x) \bmod B$ is set, $0 \leq i < k$. In a split Bloom filter, the filter is split into equal-sized non-overlapping "lanes", each of size L. Upon insertion, the bits $iL + (h_i(x) \bmod L)$ for $0 \leq i < k$ are set; in other words, a single bit is set in each lane.

When a block Bloom filter is used with block size $B = 256$ and lane size $L = 32$, it is possible to use SIMD instructions to perform the eight hash function computations at once, set the eight bits at once (one per 32-bit lane), or check those eight bits at once. The resulting Bloom filter has constant-time branch-free insert and lookup and is consistently faster than a cuckoo filter of the same size (See Figures 13 and 14 in Section 6) [26, 29, 41, 43].

Taffy block filters use split block Bloom filters as a building block to make an extensible filter with lower query time than a traditional Bloom filter would require in the same application.

2.2 Cuckoo hashing

TCFs and MTCFs are based on cuckoo hashing, a method of collision resolution in open-addressing hash tables that assigns each key a small set of slots it can occupy [47]. A cuckoo hash table consists of two arrays of size $(1+\delta)n/2$ to store a set of n keys, for $0 < \delta < 1$. The arrays are broken up into contiguous non-overlapping buckets [21, 58]. Each key is assigned one bucket per array via the application of two hash functions on the key. Every key in the table will be stored in a slot in one of those two buckets.

Inserting a key is more complex. If no slot in the two buckets for storing a key is empty, one of the occupying keys is evicted and replaced by the key being inserted. Now the victim of the eviction is in turn inserted. With high probability, eventually the evictions find an empty slot and the chain of evictions ends [47].

2.3 Succinct dictionaries with quotienting

Maps of size n with keys from a "universe" of size U can be naïvely stored in $n \lg |U|$ bits by storing every element (in any order) in an array of size n.[2] Space can be saved using a technique called "quotienting" [4, 38]. See Figure 3.

---

[2] A "universe" is the set of all possible keys, such as all 64-bit integers, or all strings up to length 1 trillion characters.
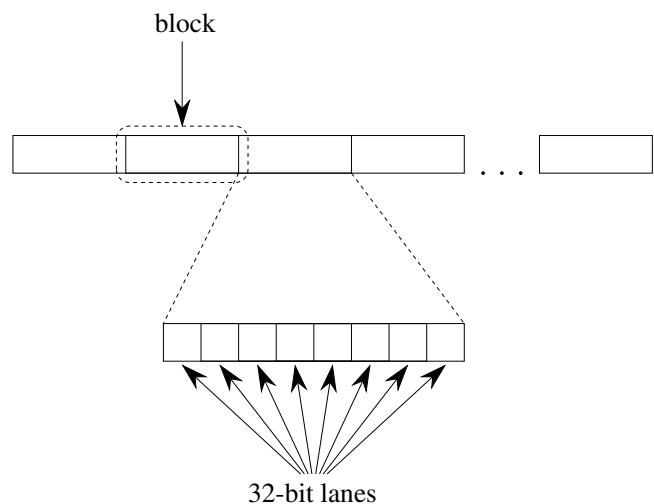

Figure 2: A diagram of a split block Bloom filter with $k = 8$ and $B = 256$.

The basic construction can be illustrated as follows: first, an array of size n is created in which each array slot can hold an arbitrary number of keys [39]. Then, a key x is stored in slot $x \bmod n$. Additionally, instead of storing x explicitly, $\lfloor x/n \rfloor$ is stored; $x \bmod n$ is the *implicitly-stored* part of they key and $\lfloor x/n \rfloor$ is the *explicitly-stored* part of the key. Because only $\lfloor x/n \rfloor$ is stored as the key, only $\lg |U| - \lfloor \lg n \rfloor$ bits are required to store it. Coming back to the array, this reduces the total storage required to $n(\lg |U| - \lfloor \lg n \rfloor)$.

In Figure 3, the column on the left represents a set of values in $\mathbb{Z}_{128}$ (integers between 0 and 127, inclusive), with each element taking 7 bits to store. The column in the middle shows another way of representing the same set as two parts per element: one of the lower order two bits and another of the higher order five. The column on the right stores the two low-order bits implicitly and the high order five bits explicitly. This cuts the space needed to store the set down from 28 bits to 20 bits.

TCFs and MTCFs use quotienting in cuckoo hashing to reduce the space needed to store the filter.

2.4 Filters that can grow

Pagh et al. describe two constructions to support extensible filters [48]. **Taffy filters refine the work of Pagh et al. with new structures for both constructions.**

$O(\lg n)$ **lookup** The first is implemented as a series of succinct dictionaries. Common similar constructions use a series of Bloom filters and exponentially decreasing false positive probabilities in each subsequent filter in order to bound the total false positive rate. That is, they create a sequence of Bloom filters with the following pairs for the false positive probability and expected number of distinct values:

$$\langle \varepsilon/2, 2 \rangle, \langle \varepsilon/4, 4 \rangle, \langle \varepsilon/8, 8 \rangle, \langle \varepsilon/16, 16 \rangle, \ldots$$

As new items arrive, they are inserted into the largest Bloom filter. Once that filter reaches the capacity it was configured for, a new Bloom filter with twice the capacity and half the false positive probability is initialized. Lookups access all the Bloom filters.

This leads to a storage footprint of more than $(\lg n + \lg(1/\varepsilon))/\ln 2$ bits per element and a query time of $O(\lg^2 n +$
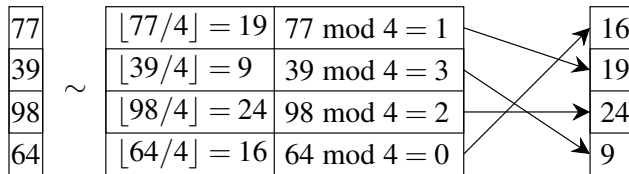
$\lg n \lg(1/\varepsilon))$. Pagh et al. reduce the lookup cost to $O(\lg n)$ by using a dictionary like Raman and Rao's that has $O(1)$ query time per filter [48, 54]. They also reduce the space usage to $O(\lg \lg n + \lg(1/\varepsilon))$ bits per element by using the sequence $\langle O(\varepsilon/i^2), 2^i \rangle$ for $i \in [1, \infty)$, rather than $\langle \varepsilon/2^i, 2^i \rangle$. See Figure 4. In this construction, $\lceil \lg(n-1) \rceil$ dictionaries are maintained with exponentially increasing capacities and logarithmically increasing bit widths. The false positive probability of the ith dictionary, counting from 1, is $6\varepsilon/i^2\pi^2$, and the sum of the false positive probabilities is $\le \varepsilon$. The lookup operation requires a dictionary lookup in $\lceil \lg(n-1) \rceil$ dictionaries.

$O(1)$ **lookup** Pagh et al. also present a filter with the same space usage but $O(1)$ query time [48]. See Figure 5.

This filter maintains a map where the keys are bit strings of length $\lceil \lg n \rceil + \lg(1/\varepsilon) + 2$ and the values (which we will call "tails") are bit strings of length up to $\lg \lg N$, where N will be the largest size of the data structure. (This definition of N is not a problem in practice, as using $|U|$, the size of the universe of keys, should be sufficient for integer keys. For non-integer keys, they must be hashed down to an integer in order to use these structures, and using the universe of the set of integers each key is hashed to also works well.) After every 2^i insertions, a new map is created where the keys are one bit longer. Pagh et al. show that the fpp of such a dictionary is no more than ε as long as $n < N$.

2.5 Compact extensible dictionaries

Hash tables that are used to accommodate sets without a size known in advance typically do so by doubling in capacity as needed. This applies to TCFs, as well. This means that at least 50% of the space goes unused at points, with an average unused percentage of at least 25%. Constructions like that of Raman and Rao are able to mitigate this, but they are largely theoretical [54]. Instead, Maier et al. use the cuckoo hashing evict operation to incrementally resize a hash table [46]. First,
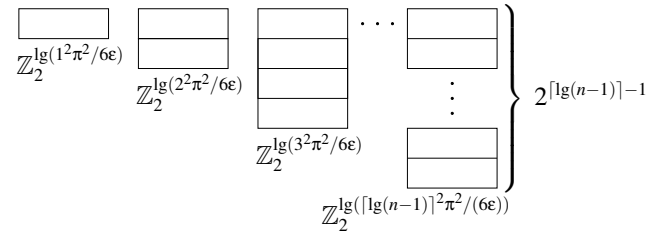

Figure 4: Pagh et al.'s first construction. $\mathbb{Z}_2^m$ means bitstrings of length m. In this diagram, columns of blocks represent dictionaries. The caption under a column is the type of the elements in the dictionary. For instance, the block with four rows in its column stores bit strings of length $\lg(3^2\pi^2/6\varepsilon)$. Quotienting (see Section 2.3) and other space factors are not presented in this figure.

Figure 3: Quotienting with $n = 4$ and all buckets holding exactly one element.

the "DySECT" table, as they call it, is broken up into equal sized sub-arrays that can be resized independently. When the table gets close to full, exactly one of the sub-arrays is doubled in size. This frees up room that's available in future eviction sequences, and the new space will slowly be filled. Eventually all arrays will have been doubled in size, thereby causing the whole table to have doubled in size without going through a phase with as low as 50% space usage.

MTCFs filters extend quotienting-based dictionaries to DySECT tables for the first time.

3 Taffy block filters

The first construction from Pagh et al. consists of a set of sub-filters of geometrically decreasing false positive probabilities but exponentially increasing size [48]; see Figure 4. As Pagh et al. describe it, this filter is initialized with a single sub-filter. Inserts take place on the most recently added sub-filter (which is the largest), while lookups are performed by performing a lookup in each sub-filter until the element is found or there are no more sub-filters to search. Once 2^i inserts have taken place, a new sub-filter is initialized and added to the collection.

Using traditional Bloom filters, the lookup cost would be

$$\sum_{i=1}^{\lg n} \lg(i^2\pi^2/(6\varepsilon)) = \sum_{i=1}^{\lg n} \lg(i^2) + \lg(\pi^2) - \lg 6 + \lg(1/\varepsilon)$$
$$= \Theta(\lg^2 n + \lg n \lg(1/\varepsilon))$$

Instead, Pagh et al. use dictionary-based filters that support constant-time lookup – such as Raman and Rao's dictionary – rather than Bloom filters [48, 54]. This reduces the lookup time to $O(\lg n)$.

Taffy block filters ("TBFs") use split block Bloom filters to keep the lookup time logarithmic and independent
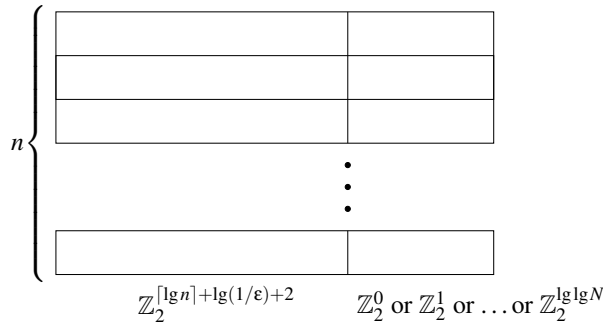

Figure 5: Pagh et al.'s second construction. In this construction of a growable filter, when a filter contains n items, it is stored as a dictionary in which the keys are bit stings of length $\lceil \lg n \rceil + \lg(1/\varepsilon) + 2$ and the values are bit strings of length up to $\lg \lg N$, where N is the largest number of keys the filter will contain.

of ε, rather than Raman and Rao's dictionary, as the latter is a theoretical, not a practical design [48, 54]. Split block Bloom filters have proven to be the fastest dynamic filters for doing single-element lookups in recent works on the matter [22,41,51]. Even though they are not based on dictionaries, they suffice for this construction, as they support the two operations needed for each level: lookup and insert. See Section 6 for performance of TBFs compared to pre-sized split block Bloom filters.

4 Taffy cuckoo filters

Taffy block filters' lookup operation requires $\lg n$ lookup operations on their sub-filters, one per sub-filter. Taffy cuckoo filters reduce lookup times to $O(1)$ and show how the ideas from quotienting can be applied to cuckoo filters to produce a dictionary.

Taffy cuckoo filters ("TCFs") use quotienting cuckoo tables to store their data, as this reduces the storage space by a significant margin (See section 2.3). See Figure 5. The keys are bit-strings of length $\lfloor \lg n \rfloor + F$ and the values are bit-strings of length up to T, for some fixed F (for "fingerprint") and T (for "tail"). By quotienting in an array of size $\Omega(n)$, each fingerprint-tail pair can be stored in $\lfloor \lg n \rfloor + F + T - \lg n + O(1)$ bits, for a total space usage of $(F+T)n + O(n)$. For performance and simplicity purposes, we pick $F + T = 15$, but this is not a requirement of the structure.

Quotienting   Quotienting is used with linear probing as the collision resolution mechanism in quotient filters [4]. Quotienting can also be used with cuckooing as the collision resolution mechanism, as in backyard cuckoo hashing [56]. Cuckoo hash tables maintain $k \geq 1$ potential locations for each key, each of which could be stored in any of its potential locations [47]. Because more than one hash function is used and because eviction occurs, it must be possible to translate from a location-element pair to an alternate location-element pair for the same key. See Section 4.1.

TCFs are based on cuckoo tables and have two arrays of slots, referred to as "sides," just as (some) cuckoo filter designs break up the address space into multiple regions, one per hash function. With TCFs, this is a requirement in order to be able to recover enough of the original key in order to re-hash it to a larger address space. TCFs, unlike backyard cuckoo hashing, use bucketing in order to increase the usable capacity and thus reduce wasted space [21,56].[3] Each side of a TCF comes equipped with a random permutation on bit-strings of length $\lg n + F - O(1)$, analogous to how each side of a cuckoo hash table comes equipped with a hash function. A fingerprint-tail pair (f,t) is stored in one of two buckets: the one in side 0 pointed to by the high-order $\lg n - O(1)$ bits of $\varphi_0(f)$ or the

---

[3]Like quotienting, buckets are not required for the correctness of the structure, just its succinctness.

```
Element := {fingerprint: ℤ₂ᶠ, tail: ∪ᵢ≤ₜℤ₂ⁱ}
Slot := Element⊥
Bucket := Slot[b]
Side(a) := {Bucket[2ᵃ], Permutation: S₂ₖ₊ₓ}
TCF(U, a) := {Side(a)[2], HashFunction: U → ℤ₂⁶⁴}
```

Listing 1: The types of a TCF.

one in side 1 pointed to by the high-order $\lg n - O(1)$ bits of $\varphi_1(f)$, where φ_i is the permutation associated with side i.

The critical part of the permutations is the ability to translate between $S_i(x)$ and $S_j(x)$ for a key x and $i \neq j$. In a cuckoo hash table in which the original key x is stored, this is trivial whether or not the hash functions associated with each side are permutations. In taffy cuckoo filters this translation is accomplished without storing x directly, but just $S_0(x)$ and $S_1(x)$, via $S_i(x) = S_i(S_j^{-1}(S_j(x)))$.

More concretely, see Listing 1. An *element* consists of two groups of bits. The fingerprint (of size F) is tested for equality when executing the lookup operation; the tail (of size $0, 1, \ldots,$ or T) is the unused part of the hashed key that will eventually be used in the fingerprint (after permuting – see below). A *bucket* consists of b possibly empty slots, each of which can hold one element or be empty.[4] A *side* consists of 2^a buckets for some a as well as a random permutation on $\mathbb{Z}_2^{a+F}$. A TCF consists of two sides and one hash function that produces a 64-bit key. The two sides have the same number of buckets but different permutations.

In Listing 1, $A \uplus B$ represents a tagged disjoint union of A and B; even if $A \subseteq B$, $A \uplus B \neq B$. $\mathtt{T}_\perp$ means the type $\mathtt{T}$ extended with the element $\perp$, indicating "null" or "empty". $\mathtt{T}[n]$ denotes an array of n values of type T. S_i is the symmetric group on $\mathbb{Z}_i$ – the set of all permutations on $\mathbb{Z}_i$. Structs are denoted by curly brackets { }.

A slot is encoded in a bitfield of size $F + T + 1$ as follows. If the last $T + 1$ bits are all zero, the slot is empty. Otherwise, there must be a one bit in the last $T + 1$ bits. All bits following that one bit are the tail. Bits $1 - F$ are the F-bit fingerprint. For example, the tail `010000` represents a tail of size 4, `0000`, while `000001` represents tail of length zero. The tail `000000` represents an empty slot, which is dinstinct from an element with a tail of length zero.

Lookup   A lookup begins by hashing a key with the TCF's hash function. Then the lookup operation does the following:

1. Applies the permutation associated with side 0, φ_0, to the most-significant $a + F$ bits in the key.

---

[4]For our implementation, we use $b = 4$. Just as with F and T, four is not a magic number, but one picked for a balance between maximum load and fpp, both of which go up as buckets get larger.

2. Reserves the next T bits of the key; this will be the key's tail. Note that these bits have not been permuted.

3. Using the most-significant a bits in the permuted bits, selects a bucket within side 0. (The remaining F bits in the permuted value are the fingerprint.)

4. Checks to see if one of the b slots in the bucket contains an identical fingerprint. If so, checks if the element's tail is a prefix of the key's tail. If yes, returns `True`. Otherwise, repeats with side 1. If neither side contains an identical fingerprint and prefix-matched tail, returns `False`.

Note that the prefix check is not strictly necessary, but does serve to reduce the fpp. See Upsize, below, as well as Figure 10 in Section 6.

Figure 6 shows how a hashed key is broken down into three parts: the index into the bucket array, the fingerprint, and the tail.

Insert   Insert places the key's fingerprint and tail in one of the $2b$ slots corresponding to that key, if an empty slot is found. Otherwise, insert selects an occupied slot from the bucket to *evict*: the element in this slot will be moved to the other side.

The evict operation first reconstructs the high order $a + F$ bits of the key by concatenating the a bits of the bucket index and the F bits of the fingerprint, then applying that side's permutation in reverse to the value. Using the same tail (this does not get permuted), the evict operation then inserts the
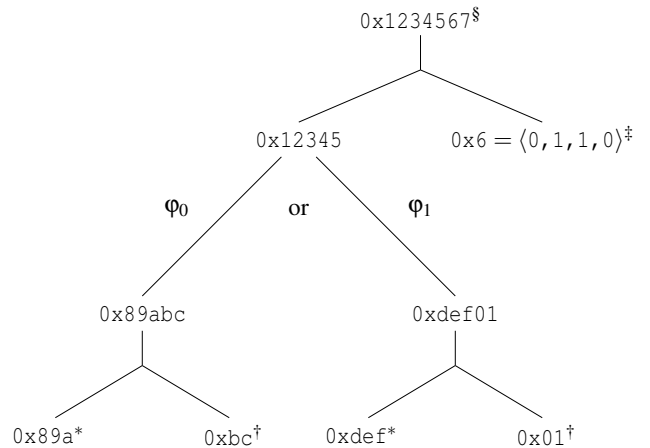

Figure 6: TCF key split. In this example, $a = 12, F = 8, T = 4$, $\varphi_0(\mathtt{0x12345}) = \mathtt{0x89abc}$, and $\varphi_1(\mathtt{0x12345}) = \mathtt{0xdef01}$.
§ hashed key
‡ tail
† fingerprint
* index into bucket array; stored implicitly using quotienting

evicted data into the opposite side; this continues until an empty slot is encountered

Upsize   When a TCF is nearly full, inserts may fail. This is identical to the situation with cuckoo filters. When this happens, the upsize method must be called to double the size of the structure. (This is not available in cuckoo filters without shortening the fingerprints, thereby doubling the fpp.)

The upsize operation begins by creating a new TCF. To transfer the data from the older to the newer TCF, upsize uses a modified version of the evict algorithm, as follows:

Upsize first reconstructs the $a + F$ bits of the key that were used to construct the bucket index and fingerprint. Then a bit is "stolen" from the tail and appended onto the end of the key. The high order bit of the tail is removed from the tail and added to the low-order end of the key. Since the tail was taken unaltered from the key, this gives $a + F + 1$ bits of the original key. The new tail has now been decreased in length by one. The key and this new tail can now be inserted into one of the sides of the new TCF as described above.

This works as long as the tail has positive length. If the tail has length zero, there is nothing to steal from. Instead, two candidate keys are created from the reverse-permuted $a + F$ bits by appending a zero and a one. It's indeterminate which one of these was in the original key, so both are inserted.

The fpp remains less than $2^{-F+O(1)}$: after adding n elements to the filter, the filter holds $n/2$ fingerprint-tail pairs with tail length $\lg\lg N$, $n/4$ pairs with tail length $\lg\lg N - 1$, ... and $n/\lg N$ pairs with tail length 1. It also contains $(\lg n - \lg\lg N)n/2\lg N$ pairs with tail length 0. Overall that's $3n/2 - O(n)$ pairs, so the space usage is still linear in the number of elements. (Note that if the tails started with length 0 instead of $\lg\lg N$, this would work out to $\Theta(n\lg n)$ rather than $3n/2 - O(n)$.) Now the odds that any bitstring of length L matches any of m different L-bit strings is $m2^{-L}$. Applying this to TCFs, since the space usage is linear, and since the "sides" (the two arrays of buckets) are of length $\lg n - O(1)$, then by reversing the quotienting operation we get that the probability that any random value that *wasn't* inserted matches with any of the existing elements is $O(n)2^{-\lg n + O(1) - F} = 2^{-F+O(1)}$; *this is the false positive probability*.

Note that if the tails all started with length 0, rather than $\lg\lg N$, then the space usage would be $\Theta(n\lg n)$ and the fpp would be $2^{-F+O(1)+\lg\lg n}$. See also [48].

Freeze and Thaw   TCFs also support *freeze* and *thaw* operations. Freeze reduces the space consumption of a TCF from $O(\lg(1/\varepsilon) + \lg\lg N)$ to $O(\lg(1/\varepsilon))$ bits per item, where N is the largest size the structure will grow to. It does so by recreating the structure as a TCF with tail length capacity 0. Thaw simply turns a frozen structure into an unfrozen structure by recreating a TCF with tail length capacity $\lg\lg N$ in which all of the tails have length zero. This allows new inserts to take place while capturing their tails.

4.1   Cuckoo filters $\cong$ cuckoo hashing with permutations and quotienting

Note that the frozen taffy cuckoo filter is a variant of a cuckoo filter in which the fingerprint hash function takes into account the index as well. In the original cuckoo filters, the two buckets a fingerprint could reside in are separated by a hashed value of the fingerprint [25]. The fingerprint stored in either bucket is identical, and there is no recovery of the original hashed value. The difference between a frozen TCF and the original cuckoo filter is that a frozen TCF can recover a prefix of the hashed key by way of inverting the relevant permutation and applying it to the bucket index and fingerprint. Other than this difference, the structures have the same operations.

This isomer of cuckoo filters shows how to support in cuckoo filters a straightforward method for porting techniques that were designed for cuckoo hash tables, including satellite data (making a cuckoo filter a type of Bloomier filter), overlapping blocks, stashes, $L > 2$ buckets, fast insertion algorithms, and cuckoo hashing with pages [12, 15, 20, 24, 28, 30, 36, 37, 42, 52, 61].

5   Minimal taffy cuckoo filters

Taffy cuckoo filters suffer from a step-function space usage: at each point, the structure has a size which is a power of two, sometimes allocating twice as much space as is needed. (See Figure 9 in Section 6.) Even if the size were not limited to being a power of two, as in vacuum filters or Morton filter, doubling the capacity during upsize would reduce the space utilization to less than 50% [8,59]. To address this, this section describes a cuckooing structure based on DySECT to reduce the space usage closer to only what is needed [46].

DySECT is a variant of cuckoo hashing. A DySECT table consists of some number of subtables, and as the table gets more and more full, it grows by doubling the size of one of its subtables. Just as in cuckoo hashing, upon an insertion, an element may be evicted. As new elements are inserted into the table, they evict older elements, and this movement causes the newly-doubled subtable to fill up.

This section proposes minimal taffy cuckoo filters ("MTCFs"), an application of the DySECT idea to quotienting and taffy filters. Some complications arise:

1. Because subtables have different sizes, the bits that are implicitly stored using quotienting vary depending on which part of the table an element is in. To address this, fingerprints in MTCFs have variable size.

2. Because fingerprints have variable size, there must be multiple permutations per side, one for each size of fingerprint.

3. Because there are multiple permutations per side, a key may be in multiple distinct buckets per side, which de-

```
Element := {fingerprint: ℤ₂^{F−1}⊎ℤ₂^F, tail: ⊎_{i≤T}ℤ₂^i}
Slot := Element⊥
Bucket := Slot[b]
Level(a) := Bucket[2][2^a] ⊎ Bucket[2][2^{a+1}]
Permutation(a) := S_{2^{p+a+F−1}}⊎S_{2^{p+a+F}}
MTCF(U, a) := {cursor: ℤ_{2^p},
               Level(a)[2^p],
               Permutation(a)[2],
               HashFunction: U → ℤ₂^{64}}
```
Listing 2: The types of an MTCF.

creases the lookup performance and increases the false positive probability.

See Listing 2 and Figure 7 for a breakdown of the components of an MTCF. In an MTCF, each element has a fingerprint of size $F-1$ or F and a tail of size up to T. A bucket consists of b (possibly empty) slots, each of which can hold one element. A level consists of two arrays of the same size, each with 2^a buckets for some a. The table consists of four permutations, one hash function, 2^p levels, and one cursor pointing to some index in the set of levels. The maximum and minimum a across all levels differ by at most 1. Levels at location less than the cursor have the larger size. If all levels have the same size, the cursor must be 0.

The permutations are grouped by side, two for each. The permutations are on values with length $p+a+F-1$ and $p+a+F$, where 2^a is the size of the smallest table, measured in buckets.

If there are larger and smaller levels, then every element in the larger levels has a fingerprint of size $F-1$, not F. This is because the implicitly-stored part of the key is one-bit longer in the larger levels, so the explicitly stored part is shorter.

In an MTCF, upsize only increases the size of one of the levels, not the whole structure. As a result, the capacity of the filter tracks more closely the number of entries in the table. (See Figure 9 in Section 6.)

Lookup A lookup operation in an MTCF first applies each of the four permutations to the hashed key.

- For the permutations on $p+a+F$ bits the first p bits indicate the level, the next a or $a+1$ indicate the bucket, and the remaining $F-1$ or F bits are the fingerprint. Lookup proceeds as it does in the TCF case, by checking if fingerprints match and if the stored tail is a prefix of the tail of the key being looked up.

- For the permutations on $p+a+F-1$ bits, the first p bits again indicate the level.

  - If the level has tables with 2^{a+1} buckets, the permuted key is not used for lookup; to do otherwise

would leave only $p+a+F-1-p-(a+1) = F-2$ bits for the fingerprint, which is not permitted. That key is simply skipped and the lookup continues with the next key.

  - Otherwise, the level has tables with 2^a buckets, and we can proceed as in the $p+a+F$ case.

Insert In insert operations, as in lookup, the first p bits of the permuted item indicate the level. Just as in TCFs, the insert operation on a bucket may produce an eviction. During an evict operation in an insert, an element may move between levels with differently-sized arrays of buckets. When the fingerprint has size $F-1$ and the level moved *from* has a bucket array of size 2^a and the level moved *to* has a bucket array of size 2^{a+1}, the number of explicitly stored bits (the fingerprint bits) is now $(a+F-1)-(a+1)=F-2$. Since every fingerprint must be of length F or $F-1$, a bit must be stolen from the tail. As in TCFs, if there are no bits to steal, two new key prefixes are created and inserted, as one of them must be the prefix of the original key.

Note that TCFs only steal bits during upsize operations, unlike MTCFs.

See Figure 8, which illustrates the transitions an element in an MTCF can go through when evicted. The states indicate the lengths of the level, fingerprint, and tail. For instance, when a level's index is less than the cursor (the center state in the diagram), the length of that level is twice what it would be if its index were higher. For elements in a short level with a short
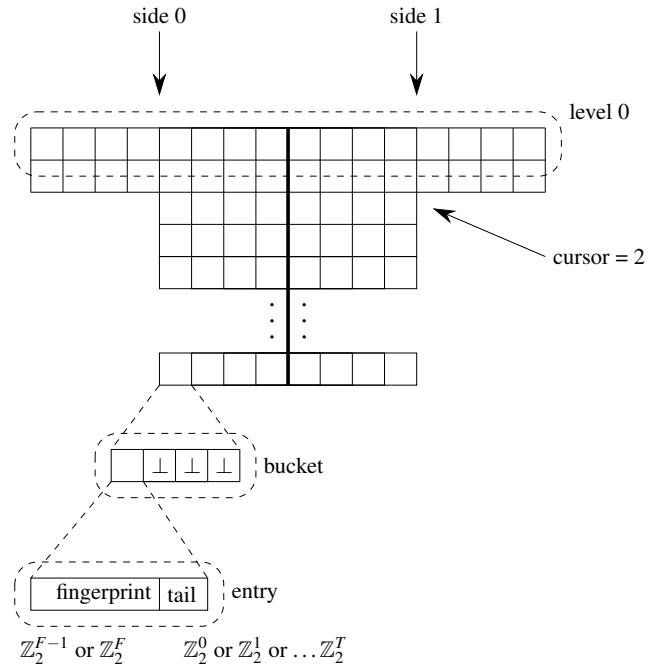

Figure 7: A diagram of an MTCF. In this example, $a = 2$ and $b = 4$.

fingerprint and a tail of length zero, when they are evicted to an element in a long level, two elements are created, as it is impossible to steal a bit from the tail of length zero.

Freeze/thaw and encoding   The analysis of freeze/thaw is identical to that of TCFs, with the exception that a frozen MTCF is not a cuckoo filter. It is, instead, a new type of filter that mixes cuckoo hashing and DySECT.

The tails are encoded as they are in TCFs - by removing the leading zero bits and the first one bit. An all-zero tail means a slot is unoccupied. The fingerprint is encoded by using a single bit to indicate if the fingerprint has size F or $F - 1$. Thus, a slot needs to store $F + 1 + T + 1$ bits. For speed of operation, we choose $F = 9$ and $T = 5$ so that slots fit into uint16_ts. As with TCFs, and for the same reason, we pick b, the number of slots in a bucket, to be 4. We pick p, the logarithm of the number of levels, to be 5; other choices are valid, but this one made a nice compromise between insert speed and space overhead.

6   Evaluation

TBFs, TCFs, and MTCFs have been implemented and tested for correctness; this section describes their space usage, false positive probabilities, and performance.

In each chart, TBFs are configured for a maximum fpp of 0.4%. All taffy filters are configured with an initial capacity of 1. All experiments were performed on both an Intel i7-7800X with 96GB of memory and SMT turned on and an AWS EC2 instance of class m6g.medium with 4GB of memory and a single Graviton2 ARM-based core. The experiments used Ubuntu 18.04 and 20.04, respectively, and g++ 10 and 9, respectively.

For performance testing, we equipped both TCFs and MTCFs with stashes, extra storage slots not associated with any bucket [37]. We set both filters to upsize when they
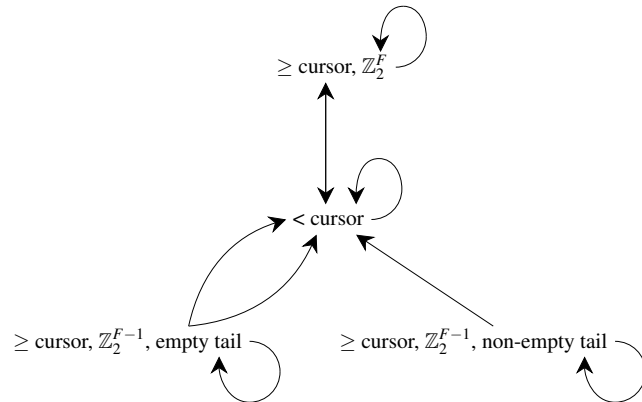

Figure 8: The transitions an element in an MTCF can go through when evicted.

were 90% full or their stashes had size greater than 4. For the random permutations we use Feistel networks with 2-independent multiply-shift as the round function [19]. These are not perfectly random, of course; analyzing the sufficiency in theory is future work. [14, 56].

For comparison, the graphs also include a cuckoo filter (labeled "CF") with fingerprints of size 12 and a split block Bloom filter (labeled "SBBF") sized to hold 100 million elements with an fpp of 0.4%. The keys used in the experiments are all randomly-generated 64-bit integers. This suffices for testing larger universes as well, as noted in Section 2.4. To benchmark the insert time, 100 million elements are inserted. At intermediate points we also benchmark the lookup operation one million times both on integer keys that are guaranteed to be present and on randomly-generated integer keys. Figures 13 and 14, which show the results of lookup performance testing, only show the randomly-generated-keys result, as the same chart for guaranteed-to-be-present keys shows the same characteristics.

6.1   Space

A filter with a false positive probability of ε must take up at least $\lg(1/\varepsilon)$ bits per element (assuming all data sets of the same size are equally likely) [10]. Practical filters use more space. For instance, Bloom filters use $\lg(1/\varepsilon)/\ln 2$ bits per element, which is about $1.44\lg(1/\varepsilon)$. Cuckoo filters and quotient filters use $(\lg(1/\varepsilon) + d)(1 + \delta)$ where d is between 2 and 3, and δ is the over-provisioning factor, between 1% - 20% [4, 25, 50]. Static filters that only support a single initializing bulk insert – such as the ribbon filter – can use nearly optimal space [22].

However, Pagh et al. showed that filters that can grow, like taffy filters can, must use at least $\lg(1/\varepsilon) + \Omega(\lg\lg n)$ bits per element [48]. Figures 9 and 10 show the space usage and ε, respectively. Cuckoo filters cannot grow (without changing the number of bits per slot and doubling the false positive rate), and as such, cuckoo filters with sufficient capacity to insert up to 100 million keys use tens of millions of bytes even when the set currently stored is very small. The same is true of split block Bloom filters. Even though the fpp of taffy filters seems to grow as the capacity grows, it is bounded above by $2^{-F+O(1)}$; see Section 4.

6.2   Time

Figures 11, 12, 13, and 14 show the performance of taffy filter operations.[5] For inserts, TBFs are the fastest of the three taffy filter variants; they are even faster than the fastest non-taffy variant, split block Bloom filters. TBF inserts are faster than the other cuckoo filters because they are simple, branch-free, and induce a single cache miss; they are faster than the pre-sized split block Bloom filter because, while being built,

---

[5] All charts of time show the minimum over nine runs.

the entirety of the TBF fits in cache until about 10 million elements have been inserted. This holds true across both tested machines, x86 and ARM.

For inserts there are visible dips in the average construction time for small filters as they get larger. These are due to measurement overhead (for the smallest *n*) and the cost of upsizing (for slightly larger *n*).

For lookups, the situation is more complex. Of the resizable filters, the taffy cuckoo filter is the fastest once the size of the filter is large enough, while a TBF is otherwise faster. The MTCF lags behind both.
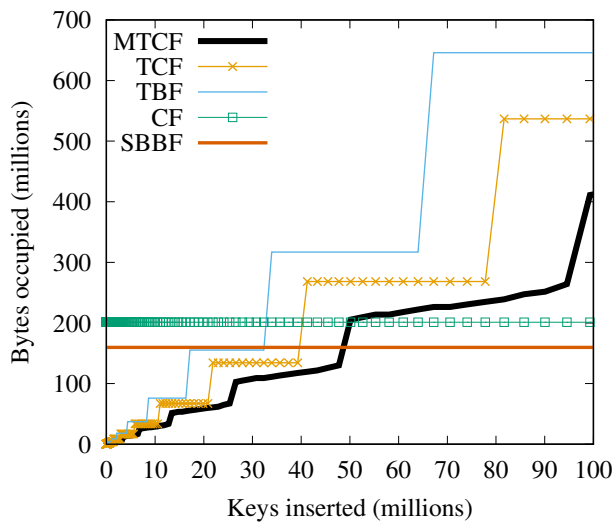
6.3 Previously-used-password filter

In this section we test the dataset of previously used passwords from "Have I Been Pwned" [35]. This dataset consists of 847 million hashes of leaked passwords. It has grown over time, starting in August 2017 with 306 million passwords. It is currently[6] on version 8.

Taffy Bloom filters and taffy cuckoo filters were tested on this dataset using the 64 low-order bits from the hashes as the keys. Both filters started out configured with an initial capacity of a single element; the TBF was configured to have a similar fpp to the TCF. Once insertion was complete, the TCF was frozen to test the lookup performance and fpp of
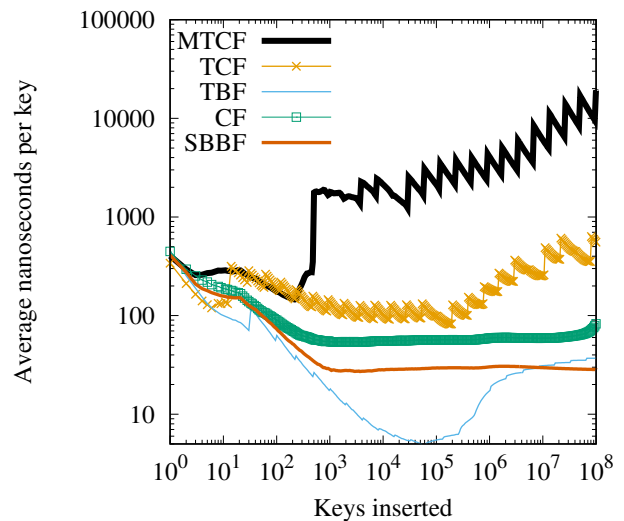
---

[6]As of December 2021

Figure 9: The amount of space used by each filter at the given number of keys inserted.

Figure 11: Insert times for filters, i7-7800X.

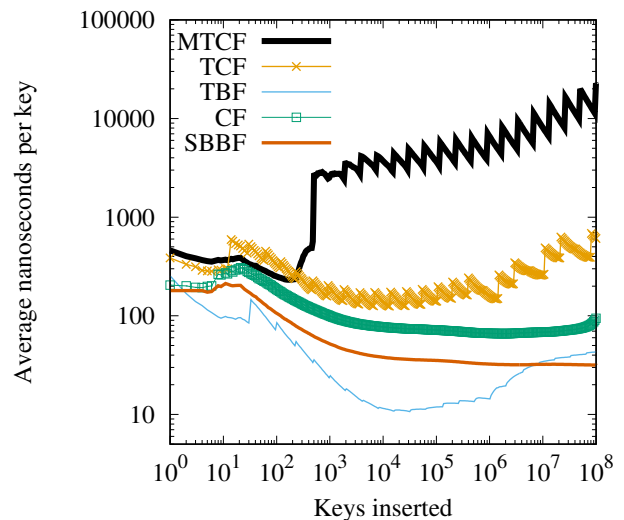Figure 10: ε, the false positive probability.

Figure 12: Insert times for filters, m6g.medium.

the resulting data structure. Experiments were conducted on an AWS EC2 r6i.xlarge with 32GiB of memory and an Intel Xeon Platinum 8375C. Times are the minimum over a set of 7 runs; fpps are the median. See Figure 15.

As in the case of the synthetic benchmarks, insert is faster for the TBF and lookup is faster for the TCF. This omission of the tails makes the false positive rate higher but the lookup faster, since the prefix checks are now unnecessary and the fingerprint matches can now be performed with SIMD-within-a-register techniques, just as in the original cuckoo filter [26]. For the "Raw, sorted" column, we consider the cost of storing 64 bits of each hash in a single array with sorting as the input method and binary search as the lookup method.
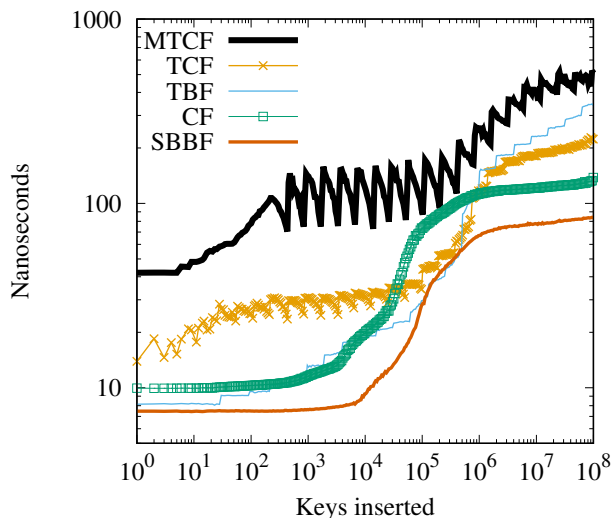

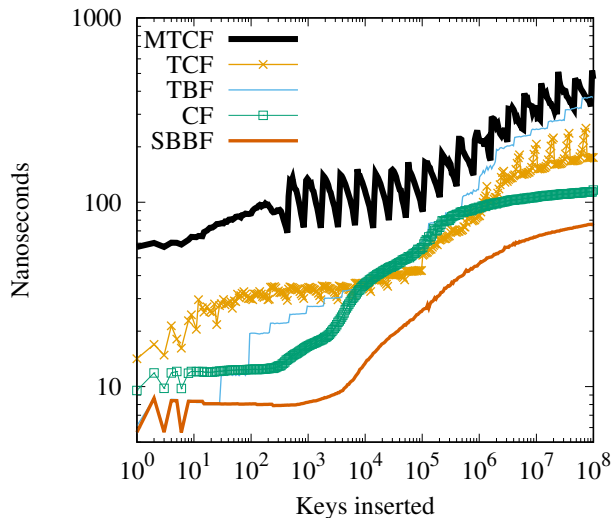Figure 13: Lookup times for filters, i7-7800X.

Figure 14: Lookup times for filters, m6g.medium.

6.4   Discussion

The MTCF offers lower space than the other two taffy filters, but its speed is substantially worse. It has significant insertion time increases when it is hard to find an eviction sequence; in this case consecutive insert operations may call upsize, causing a spike in the graph. (See Figures 11 and 12.) This cyclic behavior was noted by Maier et al. [46].

During lookup operations on MTCFs, when the cursor is close to 32, the performance improves as the four potential locations to look for a key are more frequently reduced to two, since the shorter permuted keys are no longer long enough for most of the levels in the structure. See Figures 13 and 14.

Split block Bloom filters and cuckoo filters are still attractive choices when the size of the set to be approximated is known in advance. When a growable filter is needed, the application matters quite a bit. If saving every byte matters, MTCFs are called for. If satellite data (as in a Bloomier filter) is needed, such as when using the filter in front of an LSM tree, a TCF or MTCF should be used, as TBFs do not support satellite data. Otherwise, a practitioner must ask themselves:

- Is the workload write-heavy or read-heavy? Write-heavy workloads favor TBFs over TCFs.

- Is the set likely to exceed one million elements (x86) or 1000 elements (ARM)? If yes, a TCF should be preferred.

The code for taffy filters is available on GitHubunder a permissive open-source license.[7]

7   Conclusion

This work exhibits for the first time practical structures supporting approximate membership queries and filter growth without exceeding $O(\lg(1/\varepsilon) + \lg \lg N)$ bits of space used per distinct key. We presented three structures: the TBF, the TCF, and the MTCF. We demonstrated taffy filter performance and correctness under synthetic and real-world benchmarks.

---

[7]https://github.com/jbapple/libfilter

	TBF	**TCF**	**Frozen**	**Raw, sorted**
insert (ns/key)	24	572	TCF + 2.2	113
fpp	0.25%	0.26%	0.71%	0%
lookup (ns/key)	290	108	70	719
space	4.1GiB	4.0GiB	2.5GiB	6.3GiB

Figure 15: Performance on "Have I Been Pwned"

Acknowledgments

References

[1] Mohammad Alaggan, Sébastien Gambs, and Anne-Marie Kermarrec. Blip: Non-interactive differentially-private similarity computation on Bloom filters. In Andréa W. Richa and Christian Scheideler, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 202–216, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable Bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[3] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 182–193, 2018.

[4] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, July 2012.

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

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

[7] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking*, pages 61–76, Cham, 2014. Springer International Publishing.

[8] Alex D. Breslow and Nuwan S. Jayasena. Morton filters: fast, compressed sparse cuckoo filters. *The VLDB Journal*, 29(2):731–754, 2020.

[9] Andrei Broder and Michael Mitzenmacher. Network applications of Bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[10] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark 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. Association for Computing Machinery.

[11] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen. Splitscreen: Enabling efficient, distributed malware detection. *Journal of Communications and Networks*, 13(2):187–200, 2011.

[12] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 30–39, USA, 2004. Society for Industrial and Applied Mathematics.

[13] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. The dynamic cuckoo filter. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[14] Kai-Min Chung, Michael Mitzenmacher, and Salil Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. *Theory of Computing*, 9(1):897–945, 2013.

[15] Flaviene Cristo, Eduardo Almeida, and Marco Alves. ViViD cuckoo hash: Fast cuckoo table building in SIMD. In *Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 288–299, Porto Alegre, RS, Brasil, 2019. SBC.

[16] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Niv Dayan and Moshe Twitto. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, pages 365–378, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. Longest prefix matching using Bloom filters. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 201–212, New York, NY, USA, 2003. Association for Computing Machinery.

[19] Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '96, pages 569–580, Berlin, Heidelberg, 1996. Springer-Verlag.

[20] Martin Dietzfelbinger, Michael Mitzenmacher, and Michael Rink. Cuckoo hashing with pages. In *Proceedings of the 19th European Conference on Algorithms*, ESA'11, pages 615–627, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1):47–68, 2007. Automata, Languages and Programming.

[22] Peter C. Dillinger and Stefan Walzer. Ribbon filter: practically smaller than Bloom and xor. *CoRR*, abs/2103.02515, 2021.

[23] G. Einziger and R. Friedman. TinySet–an access efficient self adjusting bloom filter construction. *IEEE/ACM Transactions on Networking*, 25(04):2295–2307, jul 2017.

[24] David Eppstein. Cuckoo Filter: Simplification and Analysis. In Rasmus Pagh, editor, *15th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2016)*, volume 53 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[25] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[26] Bin Fan, Jim Apple, Florian Jacob, Daniel Baker, Dave Andersen, José Luis Pereira, Antonio Mallia, and Arni Birgisson. Cuckoo filter. https://github.com/efficient/cuckoofilter, 2017.

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

[28] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.

[29] The Apache Software Foundation. Impala 2.7.0, 2016.

[30] Pengtao Fu, Lailong Luo, Shangsen Li, Deke Guo, Geyao Cheng, and Yun Zhou. The vertical cuckoo filters: A family of insertion-friendly sketches for online applications. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 57–67, 2021.

[31] Dimitris Geneiatakis, Nikos Vrakas, and Costas Lambrinoudakis. Utilizing Bloom filters for detecting flooding attacks against SIP based services. *Comput. Secur.*, 28(7):578–591, October 2009.

[32] Michael T. Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 792–799, 2011.

[33] Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than Bloom and cuckoo filters. *ACM J. Exp. Algorithmics*, 25, 2020.

[34] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. The dynamic Bloom filters. *IEEE Transactions on Knowledge and Data Engineering*, 22(1):120–133, 2010.

[35] Troy Hunt. Have I been pwned. https://haveibeenpwned.com/Passwords. Accessed: 2021-12-20.

[36] Megha Khosla and Avishek Anand. A faster algorithm for cuckoo insertion and bipartite matching in large graphs. *Algorithmica*, 81(9):3707–3724, 2019.

[37] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, December 2009.

[38] Donald E Knuth. *The art of computer programming: Sorting and searching*, volume 3, chapter 6.4, Exercise 13. 1973.

[39] Dominik Köppl, Simon J. Puglisi, and Rajeev Raman. Fast and Simple Compact Hashing via Bucketing. In Simone Faro and Domenico Cantone, editors, *18th International Symposium on Experimental Algorithms (SEA 2020)*, volume 160 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[40] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *In Proc. CIDR'15*, 2015.

[41] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proc. VLDB Endow.*, 12(5):502–515, January 2019.

[42] Eric Lehman and Rina Panigrahy. 3.5-way cuckoo hashing for the price of 2-and-a-bit. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, pages 671–681, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[43] Jianyuan Lu, Ying Wan, Yang Li, Chuwen Zhang, Huichen Dai, Yi Wang, Gong Zhang, and Bin Liu. Ultra-fast Bloom filters using SIMD techniques. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):953–964, 2018.

[44] Chen Luo and Michael J Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[45] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard TB Ma, Xueshan Luo, and Bangbang Ren. The consistent cuckoo filter. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 712–720, 2019.

[46] Tobias Maier, Peter Sanders, and Stefan Walzer. Dynamic space efficient hashing. *Algorithmica*, 81(8):3162–3185, 2019.

[47] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[48] Rasmus Pagh, Gil Segev, and Udi Wieder. How to approximate a set without knowing its size in advance. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 80–89, 2013.

[49] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 775–787, 2017.

[50] Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, and Rob Johnson. Vector quotient filters: Overcoming the time/space trade-off in filter design. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, pages 1386–1399, 2021.

[51] Orestis Polychroniou and Kenneth A. Ross. Vectorized Bloom filters for advanced SIMD processors. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, New York, NY, USA, 2014. Association for Computing Machinery.

[52] Ely Porat and Bar Shalem. A cuckoo hashing variant with improved memory utilization and insertion time. In *2012 Data Compression Conference*, pages 347–356, 2012.

[53] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash- and space-efficient Bloom filters. In Camil Demetrescu, editor, *Experimental Algorithms*, pages 108–121, 2007.

[54] Rajeev Raman and Satti Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Automata, Languages and Programming*, pages 357–368, 2003.

[55] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proc. VLDB Endow.*, 10(13):2037–2048, Sep 2017.

[56] G. Segev, Y. Arbitman, and M. Naor. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 787–796, 2010.

[57] Eugene H. Spafford. OPUS: Preventing weak password choices. *Computers & Security*, 11(3):273–278, 1992.

[58] Stefan Walzer. Load thresholds for cuckoo hashing with overlapping blocks. *45th International Colloquium on Automata, Languages, and Programming: ICALP 2018, Prague, Czech Republic, July 9-13, 2018*, 107:art. 102, Jul 2018.

[59] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. Vacuum filters: More space-efficient and faster replacement for Bloom and cuckoo filters. *Proc. VLDB Endow.*, 13(2):197–210, October 2019.

[60] Yuhan Wu, Jintao He, Shen Yan, Jianyu Wu, Tong Yang, Olivier Ruas, Gong Zhang, and Bin Cui. Elastic Bloom filter: Deletable and expandable filter using elastic fingerprints. *IEEE Transactions on Computers*, pages 1–1, 2021.

[61] Zhuohan Xie, Wencheng Ding, Hongya Wang, Yingyuan Xiao, and Zhenyu Liu. D-ary cuckoo filter: A space efficient data structure for set membership lookup. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 190–197, 2017.

[62] Shuiying Yu, Sijie Wu, Hanhua Chen, and Hai Jin. The entry-extensible cuckoo filter. In Xin He, En Shao, and Guangming Tan, editors, *Network and Parallel Computing*, pages 373–385, Cham, 2021. Springer International Publishing.

[63] Fan Zhang, Hanhua Chen, Hai Jin, and Pedro Reviriego. The logarithmic dynamic cuckoo filter. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 948–959, 2021.