CS 244 Final Project: Cuckoo for Filters

Walter Mostowy and Nathan Butler

ABSTRACT

In networking and other domains, it is often desired to test membership of items in very large sets without incurring large memory or performance costs. Bloom filters have long remained a popular solution to this need, but it comes with drawbacks such as poor data locality and an inability to delete items from the set. There have been many alternate approaches attempting to improve on one or more aspects of other filters, such as the counting Bloom filter, the blocked Bloom filter, the d-Left Counting Bloom filter, and the Quotient filter. More recently, Fan et al present yet another alternative, the Cuckoo filter, and show that it has notable performance benefits over its competition.

In this paper, we present our attempt to replicate some of the key findings of Fan et al in their presentation of the Cuckoo filter. We describe how we implemented six filters, tuned them to the extent they were specified by Fan et al, and tested their insertion and lookup performance. Although we find differences in our results versus those of Fan et al, we confirm that Cuckoo filters overall have excellent performance relative to their peers and speculate that the differences largely lie in hardware differences and underspecification.

In 2014, Fan et al. [5] introduced a new alternative, the cuckoo filter. Based on the cuckoo hash table, a cuckoo filter is a hash table that stores a partial fingerprint for each added element. The length of the fingerprint is controlled by the desired false-positive rate, and any particular fingerprint can be stored in one of two buckets. Set membership is tested by looking for an exact partial fingerprint, and removing an element is a simple matter of removing the corresponding partial fingerprint from the table. Fan et al. present benchmark data arguing that while cuckoo filters may have worse asymptotic performance in theory, in practice they are both more performant and use less memory than even the best Bloom-filter variants.

Our goal in this project was to reproduce some of the results from the original cuckoo filter table. In particular, we strived to reproduce Table 3, where Fan et al. compared speed, space efficiency, and accuracy of 6 different filters. They did so between a cuckoo filter, a variant of the cuckoo filter called a semi-sort cuckoo filter, a quotient filter, a bloom filter, a blocked-bloom filter, and a d-left counting bloom filter. In the course of this project, we implemented all 6 filters and attempted to reproduce that figure.

1 INTRODUCTION

In today’s era of Big Data, it is often desired to test membership of items in large sets, but the data sets themselves are, well, Big. Often too big. In many applications, these data sets are far too large to be manageably stored. This problem is especially acute in online algorithms, which are of course widely used in networking. Imagine, for example, a production web server that wanted to determine at page-load time whether an IP address-username pair had been seen before in its logs.

A widely used solution to this problem is the Bloom filter. Presented in 1970 [2], the Bloom filter has been well studied and is widely used today as a memory-efficient method of approximately testing for set membership. For example, Google Chrome uses a Bloom filter to identify malicious URLs. However, the Bloom filter has some limitations: items cannot be removed from a Bloom filter once added, and it does not work well with caches since the data-locality of Bloom filters is poor. Some variations on the Bloom filter exist but have their own shortcomings, such as the counting Bloom filter[6] that supports deletion at the expense of a much larger memory footprint.

2 BACKGROUND

There are many domains for which it is useful and desired to maintain a very large set and test elements for membership in that set. For example, Google Chrome tests URLs against a large compiled list of malicious URLs; some routers want to quickly test IP addresses for membership in a group associated with an interface; many networked applications benefit by cheaply testing for item existence in order to avoid querying a remote server about items that do not exist.

The Bloom filter [2] was an early data structure that attempted to address this need. It was quite successful and continues to see great popularity today. However, the Bloom filter has some limitations: items cannot be removed from a Bloom filter once added, and it does not work well with caches since the data-locality of Bloom filters is poor. There have been many attempts to improve on them: some aim to add the ability to delete elements from the set, others aim to reduce memory footprint, and still others attempt to improve lookup performance. While an overview of all such improvements is beyond the scope of this paper, we give a brief description of some of these attempts in this section.
2.1 Bloom Filter

A Bloom filter [2] represents a set using a large bit array and a small group of hash functions. Its API offers two functions: Insert, which adds a new element to the set, and Contains, which tests an element for membership against the set. Initially, the bit array is zeroed out. In order to add an element to the set, for each hash function, the element is hashed, the resulting hash is used as an index into the bit array, and the corresponding bit is set to 1. Testing a new element for membership is simply a matter of testing whether each of those index bits are 1. Thus, the Bloom filter allows for false positives: if any of the bits are 0, the Bloom filter can say “definitely not a member,” and otherwise, the Bloom filter will say “probably a member.”

However, the Bloom filter has some limitations, most notably that elements cannot be removed from the set once added. It is easy to see that attempt to zero out an element’s bits may corrupt other elements in the set. In addition, the data-locality of Bloom filters is poor.

2.2 Blocked Bloom Filter

The Blocked Bloom Filter [8] is an attempt to rectify the poor data locality of a standard Bloom filter. A Blocked Bloom Filter comprises an array of small Bloom filters, where each small Bloom filter is sized to fit within a single cache line. In order to add an element to the set, a single small Bloom filter is selected, and only bits within that Bloom filter are set to 1. (The filter can be selected by, for example, using the least significant bits of the first hash.) Therefore, adding or testing for membership only requires loading one cache line into memory, rather than the entire filter.

2.3 Counting Bloom Filter


In a Counting Bloom Filter, each bit of a standard Bloom filter is replaced with a counter (several bits long). Adding a set means incrementing the counters indexed by each hash; testing for set membership means checking that each indexed counter is at least 1; and deletion means decrementing each indexed counter.

The most notable downside of a Counting Bloom Filter is that its memory footprint is much larger than that of a standard Bloom filter. Typically, each counter is four bits, which means the total memory is multiplied by a factor of four.

2.4 d-Left Counting Bloom Filter

The d-Left Counting Bloom Filter [3] (dLCBF) improves on the large memory footprint of a standard Counting Bloom Filter, though it is complicated by having many tunable parameters: number of subtables, number of buckets, entries per bucket, length of fingerprint, and length of counter.

A dLCBF contains an array of hash tables, say 4. Each hash table is an array of buckets. Each table has an equal number of buckets, each bucket has an equal number of “cells” (entries), and each cell has space for a fingerprint and a counter.

Insertion of an element works as follows. The element is hashed to yield a fingerprint, which itself comprises a bucket index and a “remainder.” However, placing this value straight into the table would not let deletion work correctly. Instead, each hash table is associated with a permutation function. For each hash table, the fingerprint is run through its permutation function to yield the candidate bucket index and remainder for that hash table. The bucket with the least occupancy is selected; ties are broken to the left (hence, d-left). Once a bucket is selected, the remainder is placed into that bucket. If the remainder already exists in one of the candidate buckets, then its counter is incremented instead. Deletion then is just removing this remainder from the candidate bucket it is found in (or decrementing the counter if it is more than 1).

This works to decrease the memory footprint of a standard Counting Bloom Filter since most of the bits of a standard Counting Bloom Filter go unused.

2.5 Quotient Filter

The Quotient Filter is a recent type of filter described by Bender et al. [1] that seeks to optimize space for sequential reads. The technique itself works similar to linear probing where items are stored in a linear array and query times depend on the size of local "clusters" which are contiguous runs of elements. As a result, they work great up to load factors of around 90%, after which one begins to see a dramatic drop off in speed.

Quotient filters traditionally use very little space. They work by finding a p-bit hash (similar to a fingerprint) and splitting the hash into a quotient and a remainder. The size of these values is tunable. They use the quotient to hash to a bucket and store the remainder along with 3 bits of meta data in each bucket.

2.6 Cuckoo Filter

The cuckoo filter is the new and improved filter introduced by Fan et al [5]. The filter stores a partial-key or fingerprint for each item. Each fingerprint is called an entry. Each bucket in the table holds b entries where b is a tunable parameter.

Traditional cuckoo hashing stores full data points in its table in an array. It does insertion by using two hash functions, h1 and h2. An element can be placed into one of two indices in the table, one given by h1 the other by h2. If either of these are empty, then the element is inserted. Otherwise,
the algorithm picks one of the two candidate indices, evicts the element, and places the new element. The evicted element goes to its other index (using whichever of \( h_1 \) or \( h_2 \) was not used). If that slot is empty, we are done. Otherwise, we repeat the evict and replace policy until an empty slot is found or a threshold number of evictions is reached and the table is deemed full. Look up in the traditional hash table is as simple looking at one of the two indices for the element.

Using the fingerprint does present a challenge: how do you move an element on eviction if you are storing only the partial key? A technique allowing for dynamic insertion of partial keys was presented by [4]. The cuckoo filter uses this method. The trick is to only use one hash function. The two possible indices are computed as follows:

\[
\begin{align*}
  f &= \text{fingerprint(data)} \\
  index1 &= \text{hash(data)} \\
  index2 &= index1 \oplus \text{hash(f)}
\end{align*}
\]

In this way, given a fingerprint and one index, we can always calculate the other index by xoring the hash of the fingerprint with the index. One thing to note is that this does keep the indices close together depending on the fingerprint size because this only changes the least significant bits corresponding to the size of \( f \). For instance, if the fingerprint size is 4, the two indices are within \( 2^4 \) buckets from each other. Further discussion on optimal fingerprint size is slightly beyond the scope of this short paper but can be found at [5].

Now that we can use dynamic insert on partial-keys, insertion into the filter works the same as insertion into a traditional hash table. First, the algorithm checks if either bucket corresponding to an index has an empty entry. If so, insertion is done. Otherwise, it evicts a random entry, replaces it, and repeats the process with the evicted element's alternate index which is found by xoring the hash of the fingerprint with the original index. This process continues until an empty entry is found or a maximum iteration threshold is reached.

Lookup can be done the same way. Just look through the entries stored at the buckets corresponding to the two indices.

3 METHODOLOGY

Our goal was to replicate some of the findings of Fan et al regarding the improvements that the Cuckoo filter makes over the prior work. Specifically, we wanted to replicate the performance of the Cuckoo filter and its "semi-sorted" variant, as well as their performance comparison to the Bloom filter, the Blocked Bloom Filter, the d-Left Counting Bloom Filter, and the Quotient Filter, in their Figure 6 and Table 3.

The bulk of the work in the project came in implementing these 6 different filters efficiently and correctly. We determined that implementing each filter in C++ was most appropriate. Echoing what Fan et al did, we used CityHash as the basis of all of our hashing functions, and each filter was allowed to fill 192MiB of memory.

3.1 Bloom Filter

Because a standard Bloom Filter is merely a large bit array, it was straightforward to calculate the number of bits allowed in our 192MiB Bloom Filter: \( 192 \times 1024 \times 1024 \times 8 \). We used nine hash functions, as specified.

As described by Fan et al, we implemented an optimization of hashing. Normally, each hash function is independent, which would mean that nine hash functions would be run upon each insertion or containment query. Instead, nine hash functions are simulated by just one call to a hashing function. CityHash64 yields a 64-bit value. The first 32 bits count as the first hash, \( h_1 \), and the second 32 bits count as the second hash, \( h_2 \). From there, each hash \( h_i = h_1 + i(h_2) \).

3.2 Blocked Bloom Filter

Fan et al noted that on their test hardware, their cache-line size was 64 bytes, so we used that value as well. We utilized the functionality that C++ provides to align the dynamic memory allocation to boundaries of 64 bytes.

Since each mini Bloom filter contains 64 bytes, a 192MiB Blocked Bloom Filter would contain \( 192 \times 1024 \times 1024/64 \) mini filters. As specified by Fan et al, we used the same hash-simulation optimization as in the standard Bloom Filter, described above.

3.3 d-Left Counting Bloom Filter

Fan et al specified that they used 4 subtables, collectively containing \( 2^{23} \) buckets each consisting of four 12-bit entries. This means that the remainder and counter total 12 bits, but the relative size of the remainder and counter were not specified.

In order to maximize table occupancy, the remainder size and the counter size must be in equilibrium, so that neither is a bottleneck. If the remainder is too small, hash collisions are encouraged and insertion will more likely fail because of counter overflow; if the remainder is too large, hash collisions are less likely so counters are not utilized, making insertion more likely to fail because buckets are full.

The original d-Left Counting Bloom Filter used 4 bits for its counters (and 12 bits for its remainders). Experimentally, we found a 4-bit counter and 8-bit remainder to be a reasonable guess as to what Fan et al used. A 3-bit/9-bit breakdown would also have been very reasonable.

\(^{1}\text{https://github.com/mostowy/cs244_finalproj}\)
3.4 Quotient Filter

Quotient filters traditionally use very little space. They work by finding a $p$-bit hash (similar to a fingerprint) and splitting the hash into a quotient and a remainder. The size of these values is tunable. They use the quotient to hash to a bucket and store the remainder along with 3 bits of meta data in each bucket. So, our per bucket space usage is $r + 3$ bits where $r$ is the size in bits of the remainder. Then, the overall usage of the filter for $m$ buckets is $(r + 3) \times m$ bits.

3.5 Cuckoo Filter

For regular cuckoo filters, we can calculate the space use by calculating per bucket space use. If we define $f$ as the size of the fingerprint in bits, then each bucket will store $b \times f$ bits, giving total space usage of $(b \times f) \times m$ where $m$ is the number of buckets.

3.6 Semi-Sort Cuckoo Filter

The Semi-Sort Cuckoo filter is a variant on the regular cuckoo filter that seeks to optimize space. The basic idea is that if we force a $b$ value of 4, we can sort the entries based on the 4 least significant bits and store an index into a table and the rest of each fingerprint instead of storing all 16 of these bits. By sorting the values of the 4 most significant bits, there are only 3786 possible permutations. So, we can store an index as a 12-bit number because $2^{12} = 4096 > 3786$. This form of compression saves 1 bit per entry.

However, it does introduce the overhead of the encoding and decoding tables for the compression. In the paper by Fan et al. [5], they said this technique works only when the tables are small enough to fit into a cache. Additionally, we only start to see space improvement when the amount of data is large enough to account for the tables. The encoding table is 3786 $\times$ 16 and the decoding table is $2^{16} \times 16$ which comes out to about 135 KB of extra overhead.

4 EXPERIMENTAL SETUP

We set up the experiment as close to the paper as possible. For the data points, we precomputed data by randomly generating 64-bit unsigned integers without duplication. For the hash function, we used CityHash [7] which produces a 64-bit hash function. This hash function is the same (albeit different seeds) used for each filter type. We ran all of our experiments on a Google Cloud Compute Engine instance with 30 GB of RAM.

The experiment provided a few measures of correctness. They are as follows:

- **Total Elements Scanned**: For our input we precomputed $2^{27}$ keys. We limited the size of each filter to 192 MB so this measured how many keys we could successfully store.

- **Space Efficiency**: Calculated by taking the total number of elements scanned divided by the space in bits required by the filter.

- **False Positive Rate**: Measure of the percent of false positives when queried with elements not seen in the key set.

- **Processing Speed**: Measure of how many operations per second is done. In the paper, their table 3 (which we are attempting to reproduce) uses only insert speed so we will talk about that here.

Each filter in turn had specific tunable parameters. Some were specified in the original paper, some were not. For our experiments in this paper, the parameters were as follows:

- **Cuckoo Filter (CF)**: For the cuckoo filter, we used the specifications from the original paper. We used a fingerprint size of 12 and $m = 2^{25}$.

- **Semi Sort Cuckoo Filter (SS)**: We used the same specifications for the Semi Sort optimization that we did for the cuckoo filter.

- **Quotient Filter (QF)**: For the Quotient filter, we also used the specifications from the original paper. We used a remainder size $r$ of 9 bits and $m = 2^{27}$.

- **Bloom Filter (BF)**: We used 9 simulated hash functions and $192 \times 1024 \times 1024 \times 8$ bits for the underlying array, as specified in the original paper.

- **Blocked Bloom Filter (BB)**: We used 9 simulated hash functions and sub-filters of 64 bytes, totaling the same number of bits as the Bloom filter, as specified in the original paper.

- **d-left Counting Bloom Filter (DL)**: The original paper specified using 4 subtables, each containing $2^{25}$ buckets, each containing four 12-bit entries. However, the length of the remainder and counter in each entry was unspecified. We decided upon 8-bit remainders and 4-bit counters; see the Methodology section for more details.

5 RESULTS

For our results, we ran 5 trials and averaged across the trials. You can see the results in 1. Our results show that overall the Cuckoo filter ran with the best stats across the board. It was able to process all the elements, had the lowest false positive rate, second fastest insert speed, and tied for best bits per item. The semi-sort variant also performed well however was surprisingly slow. The d-left counting filter ran the fastest however was not able to process all the elements like the other filters were and had the worst space efficiency. The bloom filter and the blocked bloom both processed all the elements but at the cost of speed in the case of the bloom filter and accuracy for both filters.
6 DISCUSSION

6.1 Comparison to target paper
Our results did not match Table 3 from the original document. We believe there are a few reasons.

First, our machines were different. They ran their tests on two Xeon processors with an L3 cache and 32 GB of DRAM. While we were able to match the amount of memory, our speeds definitely seemed to be affected by not having the computational power. We also did not have the L3 cache that their machine did. The cache in particular was important for the Semi-Sort Cuckoo Filter because of the extra overhead of the encoding and decoding table. If that is in the cache, index lookups become much faster.

Second, our stopping conditions must have been different. We were able to process all the elements in 4 of the 6 filters whereas the original paper did not filter all the elements with any of the filters. While we matched the total number of elements that we were trying to filter, we obviously did not stop the same time as the paper we were trying to match. Our stopping conditions for the cuckoo filters were when we did 500 iterations of evict and replace without finding an empty bucket and for the quotient filter and bloom filters when we hit a 90% load. Our stopping conditions being different affected a few different parameters besides total number of elements filtered. It also made our bits per item be off as well. Because all filters were 192 MB, if they filter all the elements, their bits per item will be the same. This causes our results to differ from the original paper. Also, having more items we predict helped drive up the false positive rate.

6.2 Limitations and Weaknesses
Our implementations were not perfect. We definitely had higher than expected false positive rates as compared the original paper.

This could be because we fundamentally got the algorithms wrong. However, we did do extensive unit testing and sanity checks so we believe that our errors did not come from a misunderstanding of how the filter works.

This could also be from the way we are testing. The testing script was the only part of the implementation that was not following a set algorithm. Differences in how we generated random values and checked false positive rates could account for the differences in our results.

Another possible source of error comes from optimizations. We spent a large amount of time optimizing to make sure our filters had no wasted bits or memory. In the course of doing so, we may have inadvertently introduced bugs and corner cases that our unit tests did not account for. We also (thinking optimistically) could have over-optimized and the results of our improved numbers for number of items filtered and bits per item could be from a more space efficient filter.

One variable that we were unable to account for was the percent of false queries in the false positive test. This was a tunable parameter that they passed in and was not discussed in the paper itself. We decided to use 50% as our parameter. That is, for the false positive test, 50% of queries were false. We of course accounted for this when calculating the percent of false positives, however, varying this parameter could have drastically affected the results.

Lastly, because we had different stopping conditions and most filters processed all the elements, this did not allow us to fully examine the differences in space efficiency and bits per item. The other paper had clear differences and was able to examine these differences. While we were able to see differences in false positive rates and insert speeds, our analysis of space efficiency is somewhat limited.

6.3 Takeaways
What the reader should take away from our attempted reproduction is that the differences in filters definitely come with tradeoffs. Taking the high speed we observed for d-left filters comes at the expense of accuracy and space. Using the Semi-Sort cuckoo filter we observed gives great space efficiency but comes at the slight cost of speed. In general, it is good to consider all the possible tradeoffs of a filter before implementing it in your system.

The reader should also take away that differences in machines used have a large effect on the performance filters. This is shown through the difference in our system as compared to the one we were attempting to recreate. The speeds were much greater in the other machine due to processor power and an improved cache.
7 FUTURE WORK

For future work, we would like to explore the differences in the filters more. In particular, this recreation of the table only represents a test done on a specific machine. We do think that the type of processor and memory of the machine that is used for the test will have a big result on the final outcome. For instance, even though the Quotient Filter performed poorly in this experiment, the Filter itself is optimized for under-provisioned systems memory wise. In other words, it was designed to limit disk accesses when memory is tight. So, naturally it would not perform well when there is 30 GB of RAM however might outperform the other filters if there was less memory.

Another avenue of exploration could come in exploring the strength of hash functions CityHash takes 20 ns to do a hash. While this is not a large amount of time, every element inserted must be hashed, so using a weaker, faster hash function might produce faster results. However, using a weaker hash function would probably affect accuracy due to higher rates of hash collisions.

REFERENCES


