Design and Evaluation of Cascading Cuckoo Filters for Zero-False-Positive Membership Services

Sai Medury
Computer Science and Engineering
University of Tennessee at Chattanooga
Chattanooga, TN
sai-medury@mocs.utc.edu

Amani Altarawneh
Computer Science and Engineering
University of Tennessee at Chattanooga
Chattanooga, TN
jwh247@mocs.utc.edu

Anthony Skjellum
SimCenter &
Computer Science and Engineering
University of Tennessee at Chattanooga
Chattanooga, TN
tony-skjellum@utc.edu

2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC) | 978-1-6654-1490-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/CCWC51732.2021.9376160

***Abstract*—The approximate set-membership data structures (ASMDS), like the Bloom filter and cuckoo filter, provide constant-time testing of set-membership. They produce false positives because of a loss of bits during compression. However, in case all potential false positives are known (or can be evaluated), it is possible to use filter cascades and collectively eliminate such false positives. The application of filter cascading algorithm to the Bloom filter was originally proposed for optimizing memory usage and is currently an integral part of CRLLite.**

Recently proposed cuckoo filters function similarly to Bloom filters but with cuckoo hashing techniques. They produce comparatively lower storage overheads and additionally support efficient deletions. Therefore, applying the cascading algorithms to the cuckoo filter will also produce lower storage overheads in comparison to cascading Bloom filters. Further, cuckoo filter's support for deletions enable efficient updates to the filter cascades.

In this paper, we present the design and analysis of *cascading cuckoo filters*, a potentially more space-optimal ASMDS in comparison to cascading Bloom filters. A novel contribution of this paper is the application of filter cascading algorithm to cuckoo filter, which has not been proposed before to the best of our knowledge.

***Index Terms*—Cuckoo filter, Approximate Set-Membership data structures, Bloom filter, Cuckoo Hashing, Data Compaction and Compression**

I. Introduction

The approximate set-membership data structures (ASMDSs) [1] provide constant-time testing of set membership. They are constant in size and can be one- or two-dimensional arrays of bits or data buckets. ASMDSs provide an approximate representation of a set of elements, and are built using data compression and data compaction techniques. They produce false positives because of lossy compression [2].

Consider a set R of elements, where $R \subset U$. The ASMDSs can help test membership of an arbitrary element e where $e \in U$. The size m of an ASMDS is typically constant and much smaller than $|R|$. The false positive rate p is dependent on the ASMDS's size and occupancy. The rate of false positives can be minimized by choosing the appropriate size of the filter (m) according to the size of the set ($|R|$) [1], but it is impossible to eliminate false positives if the domain (U) of element e grows arbitrarily large [3]. However, in case of a *finite* set U, where all potential false positives are known (or can be evaluated), it is possible to use filter cascades and collectively eliminate false positives [3].

The Bloom filter [2]—considered to be the canonical example among ASMDSs [1]—is a one-dimensional bit array and uses k hash functions to set an (almost) unique combination of bits according to given the input element. Other proposed variants [4]–[7] of Bloom filter work to improve spatial efficiency and/or support deletion of elements from the set. The application of a filter cascading algorithm to Bloom filter was initially proposed by [8] for the purpose of optimizing memory usage. To the best of our knowledge, it is also the first research work to definitively eliminate false positives in Bloom filters, although under limited preconditions. It is also currently being implemented as an integral part of CRLLite [3], a revocation information distribution system used in Mozilla Firefox.

The recently proposed cuckoo filter [9] functions similarly to Bloom filters but with cuckoo hashing techniques. They produce comparatively low space overhead and additionally support efficient deletions. Therefore, applying a cascading algorithm to cuckoo filter would logically produce lower spactial overhead as compared compared to cascading Bloom filter. Further, a cuckoo filter supports deletions, which makes it more efficient to update the filter cascades.

In this paper, we present the design and analysis of *cascading cuckoo filters*, a potentially more space-optimal ASMDSs in comparison to cascading Bloom filters. A novel contribution of this paper is the application of filter cascading algorithm [8] to cuckoo filter, which has not been proposed before to the best of our knowledge.

The remainder of this paper is organized as follows: in Section II, the concepts and properties of Bloom filter and cuckoo filter are described in detail, including a comparison between them. The filter cascading algorithm and how they eliminate false positives is also explained in the same section. In Section III, our novel design for cascading cuckoo filters is introduced and discussed, including a description of each operation involved and their space and time complexities. In Section IV, we discuss collision handling in cascading cuckoo filter and provide a comparison of computational complexities between cascading cuckoo filter and cascading Bloom filter. We present our planned future work in Section V, then we conclude with a brief summary in Section VI.

978-0-7381-4394-1/21/$31.00 ©2021 IEEE

II. Background

The Bloom filter is an approximate member query data structure that guarantees true negative results but cannot guarantee true positive results. In other words, the Bloom filter is a probabilistic data structure where there is a probability that a particular element is present in the list but not for sure [10]. As a constant-sized bit-array, the Bloom filter requires less storage than other similar data structures like hash table. It provides constant-time set-membership testing regardless of size of the set, which makes it highly beneficial. A Bloom filter has four main characteristics; 1) the number of bits in the filter or the length of the bit array (one dimensional array), 2) the number of the hash functions, 3) the number of items that are in the filter, and 4) the calculated probability of the false positives. It is important to carefully choose the first three characteristics to minimize probabilities of getting false positive results. These characteristics cannot be modified once the filter is instantiated. As proven by [11], the rigorous upper bound ϵ for false positives for a finite Bloom filter with m (m¿1) bits, n elements, and k hash functions is at most:

$$\epsilon \leq \left(1 - e^{-k(n+0.5)/m-1}\right)^k \tag{1}$$

Another space-efficient probabilistic data structure is a cuckoo filter, as mentioned above. It is similar to the Bloom filter, but with a lower false positive rate, and with a zero false negative rate like a Bloom filter [12]. A cuckoo filter uses a hash table employs independent hash functions; each hash function determines the row of the fingerprint of an element within that table. Insertion of an element to a row that already has one (a collision) will cause that element to move (termed *kick*) the existing element's fingerprint to the corresponding position determined by the second hash function. If that position is already filled, then the existing element will be kicked out. This process will be repeated until either the new element is successfully inserted, or the insertion algorithm reaches a set number of iterations (termed *maximum number of kicks*) that indicates an infinite loop. In this case, the cuckoo filter rehashes and finds all new hash functions and reinserts all of the elements [9].

The cuckoo filter allows hashing of an element into a number of bits, called *fingerprint*, to identify the bucket index in which this element will be stored. It also allows more than one element candidates to be stored in the same index [9]. Storing the fingerprint of an element instead of setting a combination of bits (like in Bloom filter) enables cuckoo filter to support efficient deletion of elements for a set [9]. Furthermore, the cuckoo filter produces less false positive rate and provides faster lookup speeds than the equivalent Bloom filter for the same input set.

Cascading algorithm originally proposed to optimize memory usage in de Bruijn graphs [13] The de Bruijn graph is a directed graph that represents overlaps between sequences of symbols. The cascading Bloom filter was used to identify potential next candidates in the graph by inserting elements into the Bloom filter cascades. If the set of potential false positives is known or finite, then it is possible to be eliminate them [8].

III. Design of Cascading Cuckoo Filter

In this section, we present the design and methodology of cascading cuckoo filter, which is the main contribution of this paper.

The *insert* and *lookup* operations provided by the standard Bloom filter [2] and cuckoo filter [9] are indistinguishable by design. Therefore, we observed that the filter cascading algorithm provided by [8], [14] can be applied to cuckoo filter without making any changes. (This filter cascading algorithm will be optimized in future work by leveraging the delete operation, which is specially supported by cuckoo filter.) For the sake of completeness, we briefly summarize the filter cascading algorithm here.

A. Precondition for successful elimination of false positives

As mentioned in Section I, it is possible to eliminate false-positives in ASMDSs only if the set of potential false positives is known or can be identified. For practical implementations, the false positives need to be identified in no more than polynomial time. This precondition also implies that the set of candidate elements for lookup (S) must be finite.

B. Cascading Algorithm

The core objective of filter cascading algorithm is incrementally to represent smaller sets of false positives in subsequent layers or "*cascades*" of primary ASMDSs. This process is expected to definitely terminate with final cascade of ASMDSs producing no false positives [3]. As shown in Fig. 1, given a set R, where $R \subset U$ and $R \cup S = U$, insertion into cascading cuckoo filter will begin with primary cuckoo filter (CF_1) that represents the set R. The CF_1 would naturally produce false positives (Set FP_1) that belong to Set S. If the precondition is satisfied, it is possible to identify all the elements in Set FP_1 by conducting a lookup operation of all elements in Set S against the primary cuckoo filter CF_1. All elements in Set FP_1 can then be inserted into a separate cuckoo filter (CF_2). Note, the CF_2 is the first level cascade of the primary cuckoo filter and it represents the (false positive) elements in S, therefore reversing the meaning of a successful lookup in CF_2. This is true for all cuckoo filter cascades at even-number layers.

The cascading process in the insertion operation is expected to continue similarly for $h+1$ rounds until no more false positives can be identified. Cuckoo filters produce a false positive rate p (less than 3% under optimized configuration), hence the size of set FP_1 is $p \times |U|$. And the size of sets represented by subsequent cuckoo filter cascades reduces by p times. The cuckoo filter configuration parameters can be further optimized according to the size of U, to reduce false positive rate and to improve spatial efficiency [9]. Similarly, the number of filters required and the overall spatial efficiency of cascading cuckoo filter can be optimized by adjusting configuration parameters. Methodology for selection is described in [9].

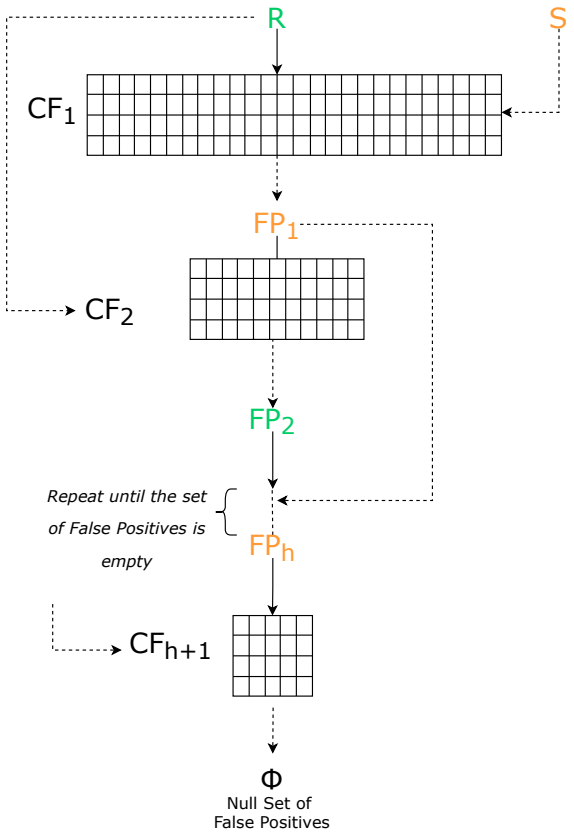

Fig. 1. Inserting elements of a set R, where $R \subset S$, into cascading cuckoo filter. Cuckoo filter at level h is represented as CF_h and the resulting set of false positives at level h is represented as FP_h. The dashed arrows represent "lookup" operations and the solid arrows represent "insert" operations.

C. Set-Membership Testing in a Cascading Cuckoo Filter

The primary cuckoo filter and its subsequent cascades represent a set of elements either belonging to R or S. An arbitrary input element e, where $e \in U$, can be used as input to the lookup operation to the primary cuckoo filter and its cascades. Note, it is important that the input element e belong to Set U for successful membership testing. The ultimate output of true or false can be produced even if $e \notin U$, but this output may not be meaningful.

Like Bloom filters, cuckoo filters do not produce false negatives. Therefore, observing a negative output to the lookup operation on any of the cuckoo filters terminates set-membership testing. Assuming no negative output occurs for any of the cuckoo filters, then the number of layers decides the result of set-membership test. As shown in Fig. 2, to test set membership of an arbitrary input element e, it is given as input to "*lookup*" operation in cuckoo filters at each level, beginning with the primary one. Unlike typical ASMDSs with a binary output of yes or no, the cascading cuckoo filter produces a multivariate output where the number of layers at which testing terminated is a deciding factor. The set-membership test output is summarized in Table I and is considered positive ($e \in R$) if

- the testing terminated with a negative output of the lookup operation at layer i and i is even; or,
- the testing terminated with a positive output on the final layer cuckoo filter (in other words, no cuckoo filter produced a negative for input element) and the total number of layers is odd.

Otherwise, the test was unsuccessful.

layer of termination	even	odd
negative output at layer i	R	S
positive output at final layer l	S	R

TABLE I
SUMMARY OF MULTIVARIATE OUTPUT OF SET-MEMBERSHIP TESTING IN CASCADING CUCKOO FILTER, SHOWING IF $e \in R$ OR IF $e \in S$

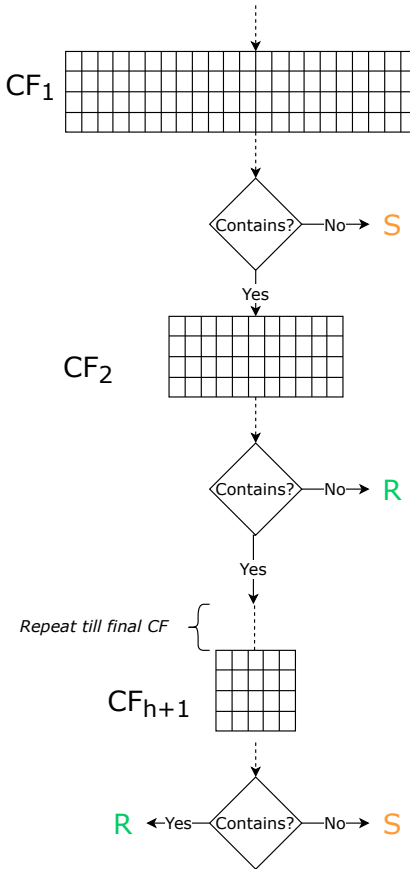

Fig. 2. Testing set membership using cascading cuckoo filters. The input element e is looked up on the cuckoo filter at each layer until either a negative output is encountered or until the final layer is reached.

D. Space And Time Complexity

Cuckoo filter, like all ASMDSs, is of constant-size regardless of the size of the set represents. However, cascading cuckoo filters are of varying size and may have up to $h + 1$ filters, where $h > 1$. The amount of space occupied by a cascading cuckoo filter is proportional to h. Optimizing configuration parameters of a cuckoo filter can help reduce

false-positive rate of the filter at each layer individually. It may also collectively reduce the number of layers in, and total space occupied by, cascading cuckoo filters.

Computations involved with the insert operation in a cascading cuckoo filter can be considered as a collection of insertion operations and search operations at each layer. The largest computation occurs in preparing the first cascade layer, where all the elements in Set S must be looked up in the primary cuckoo filter (CF_1) to determine the set of false positives (FP_1). This operation alone requires computation in the order of $O(|U|)$. Computations required for subsequent layers reduces incrementally by a magnitude determined by the false-positive rate at each layer. The overall computational time complexity of preparing cascading cuckoo filters is proportional to size of superset U, and is of the order $O(|U|)$. Note that the cascading cuckoo filter ultimately represents the set R, which is a small subset of U, and can provide set-membership definitively only for elements in R. It is important further to note that computational time-complexity of insertion in a cuckoo filter is in fact proportional to the maximum number of *kicks* allowed during insertion and only its amortized time is of $O(1)$ [9]. The maximum number of kicks, regardless of how frequently it is reached, can have significant impact for large sizes of set R.

The computational complexity of set-membership testing is also proportional to the number of layers h in cascading cuckoo filter. Nevertheless, it is close to constant-time since the value of h is small and doesn't vary significantly with variation in the cardinality of sets.

IV. Analysis

In this section, we discuss collision handling in cascading cuckoo filters and analyze them in comparison with cascading Bloom filters. Potential areas for optimization are also noted and described.

A. Collision Handling

The hash functions used in a cuckoo filter are independent but not randomized. In theory, they produce collisions with high probability [15]. However, an empirical evaluation of cuckoo filters reveals otherwise, as described in [9]. If the size of buckets and fingerprint size are both optimized, practical applications of cuckoo filter are able to achieve minimized false-positive rates ($< 5\%$) while maintaining maximum occupancy ($> 95\%$) [9]. By extension, the configuration parameters in cascading cuckoo filters must also be tweaked according to input elements. These application-specific customizations are necessary to maintain space-efficiency. They are unavoidable and must be the first step in building any practical application.

B. Comparison with the Cascading Bloom Filter

The Bloom filter requires 44% more space than the cuckoo filter for a target false positive rate below 3% [9]. So the primary cuckoo filter in cascades will naturally occupy less space. The difference in space will multiply with each layer. Therefore, the overall space occupied by cascading cuckoo filter will also be comparatively lower than the equivalent cascading Bloom filter. In some cases, this difference may also result in a lower number of layers.

But, the number of *kick* operations in cuckoo filter during insertion can cause significant overhead, especially in the presence of multiple filters. This contrasts with a Bloom filter, insertion is truly of constant time computational complexity, making the cascading Bloom filters slightly more efficient than cuckoo filters for insertions.

The cascading Bloom filter only supports batch construction and it requires reconstruction after every deletion in the input set (R) or after any update in the complementary subset (S). This can be avoided in cascading cuckoo filter since cuckoo filter supports deletions. Furthermore, cuckoo filter's support for efficient deletions may enable cascading cuckoo filters to support individual updates. This will be either proven or disproven in future work through further study and analysis.

V. Future Work

Evaluating the relationship between cuckoo filter parameters, the size of input sets, and number of cascades is an interesting subject for future work. Simulations may reveal lower bounds and upper bounds on space complexities that may prove valuable while selecting computation platform for application. Prototyping the cascading cuckoo filter and comparing the performance and space efficiency through benchmarking will provide insights into the difference in overhead between cascading cuckoo filter and cascading Bloom filter. Empirical analysis of collision handling will show the true probability of collision occurrences, which is intuitively bound to be higher in cascading cuckoo filters than it is in a single cuckoo filter. Generalizing the filter cascading algorithm will unlock the potential for other current and future ASMDSs to definitively eliminate false positives.

Managing deletions in the cascading cuckoo filter will be a next step in our research as well. Single cuckoo filters can do so. However, to achieve deletion in a cascading filter, without rebuilding the entire data structure, requires careful thought. As collisions disappear, it would appear possible to delete layers, for example. Considering sets of additions and deletions in groups might be more optimal in such situations to provide optimal transformations of the filter as sets grow and shrink.

VI. Conclusion

In this paper, we presented the design and analysis of *cascading cuckoo filters*, a potentially more space-optimal ASMDSs in comparison to cascading Bloom filters. A novel contribution of this paper is the application of filter cascading algorithm to cuckoo filter, which has not been proposed before to the best of our knowledge.

We build on the observation that the application of cascading filters is independent of whether the underlying filter is Bloom or another filter. So, we straightforwardly showed that a cascading cuckoo filter is feasible. However, since a

cuckoo filter has superior spatial properties for a given false-positive rate, we observed that the cascading cuckoo filter would be more efficient in terms of space. We noted how multiple levels of the cascading algorithm could be created to deal with collisons arising at insertions. We indicated that the Bloom filter would be more efficient for insertions, given their constant-time property. We left managing the deletion function in our cascading cuckoo filter–available in the cuckoo filter but not Bloom filters—as future work.

VII. ACKNOWLEDGEMENTS

Research reported in this publication was supported in part by the National Science Foundation under grants nos. 1812404 and 1821926. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Alex D. Breslow and Nuwan S. Jayasena. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proc. VLDB Endow.*, 11(9):1041–1055, May 2018.

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

[3] James Larisch, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. Crlite: A scalable system for pushing all tls revocations to all browsers. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 539–556. IEEE, 2017.

[4] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms*, pages 684–695. Springer, 2006.

[5] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293, 2000.

[6] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.

[7] Afton Geil, Martin Farach-Colton, and John D Owens. Quotient filters: Approximate membership queries on the gpu. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–462. IEEE, 2018.

[8] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading bloom filters to improve the memory usage for de brujin graphs. *Algorithms for Molecular Biology*, 9(1):2, 2014.

[9] 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. ACM, 2014.

[10] Shahabeddin Geravand and Mahmood Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, 57(18):4047–4064, 2013.

[11] Ashish Goel and Pankaj Gupta. Small subset queries and bloom filters using ternary associative memories, with applications. *ACM SIGMETRICS Performance Evaluation Review*, 38(1):143–154, 2010.

[12] Wikipedia contributors. Cuckoo filter — Wikipedia, the free encyclopedia, 2020. [Online; accessed 30-December-2020].

[13] Wikipedia contributors. De bruijn graph — Wikipedia, the free encyclopedia, 2020. [Online; accessed 31-December-2020].

[14] 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*, pages 30–39. Citeseer, 2004.

[15] Alex Chumbley and Christopher Williams. Cuckoo filter. https://brilliant.org/wiki/cuckoo-filter/. [Online, Accessed:8/15/2018].