ITEM-PERSONALIZED BLOOM FILTERS

Krishnan Venkataraman, Kavya Vaddadi
{kvenkat9, kvaddad1}@jhu.edu

ABSTRACT

Probabilistic data structures are effective in dealing with approximate membership querying, specifically while dealing with extremely large data-sets. Bloom filters are one such popular probabilistic data structures which can determine whether an item is definitely not present or probably present in processing larger data. A large collection of variants of bloom filters emerged over-time specifically with a target of improving upon the probabilistic aspect of false positivity while optimizing several application specific operations. In this work, we propose an extension to conventional bloom filter with an objective of verifying the improved space-efficiency while mitigating the false positivity under diverse experimental conditions. We introduce a notion of item-personalized insertions in the context of traditional bloom filters which we refer as Item-personalized bloom filters (IBF). Our initial empirical experimental evaluations show that IBF indeed helps to marginally reduce false positive rate while improvising on the space-efficiency. Futuristically, we like to explore extensive randomized experiments to stabilize and extend IBF in diverse conditions on large filter variants.

***K*eywords** Standard bloom filter · Item-personalized bloom filter · Re-hashing

1 Introduction

Probabilistic data structures are particularly effective when dealing with large data sets. While dealing with the data, most of the time, one would need to execute a simple check to verify *"whether an item is already present or not"* while processing the real-time data. The conventional method for such queries is to use a HashMap or a HashTable, or to store the data in an external cache. However, the difficulty is that these simple data structures can't fit into memory with enormous data sets. Because of their space and time advantages, probabilistic data structures are useful in this situation to determine whether an item is definitely not or possibly might be present, which is popularly referred to as approximate membership querying.

Approximate mmbership querying has a wide range of applications in several large scale data-based applications. For instance in genome-based studies, specifically while dealing with large-scale metagenome data analysis [1].

2 Background and Explanation

Several popular variants of bloom filters are in use to cater different applications and improvements over each other in terms of functional or performance aspects. The most popular variants that emerged from conventional or Standard bloom filters include Counting bloom filters, Blocked bloom filters, Cuckoo filters and their implementation/performance sub-variants. Below, we provide a quick overview on individual categories, and also provide a brief summary on the latest state-of-the-art sub-variants that are introduced for handling a specific criteria.

Standard Bloom Filters [2] (SBF):
A Bloom filter consists of k hash functions and a bit array with all bits initially set to "0". To insert an item, it hashes this item to k positions in the bit array by k hash functions, and then sets all k bits to "1". Lookup is processed similarly, except it reads k corresponding bits in the array: if all the bits are set, the query returns true; otherwise it returns false. This form of standard bloom filters do not support deletion. Figure 1 shows the schematic representation of SBF.

Counting Bloom Filters [3] (CBF):
CBF can be considered as an immediate extension of SBF. SBF is specially extended by CBF to allow deletions. In

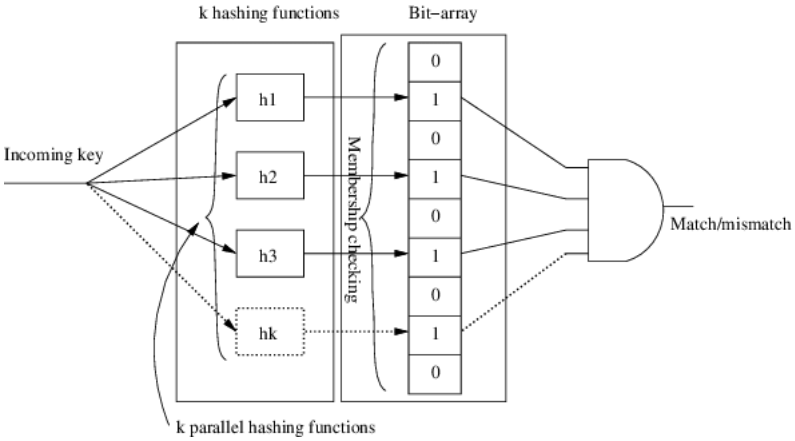

Figure 1: A standard bloom filter with k hash functions setting corresponding k bits in the Bit-array.

place of an array of bits, CBF utilizes an array of counters (incremental integers). Instead of merely changing k bits, an insert increments the value of k counters, and a lookup verifies that each of the required counters is non-zero. These k counters' values are decremented by the delete action. Several implementation variants [] are worked around the idea to retain the SBF properties by maintaining the array in CBF to be large enough for accommodating larger incrementally feasible values. Hence, CBF in practice holds up at least 4 times more space than SBF (integer storage instead of single bit collection). Additionally there are space-efficient implementations of CBF which considers secondary/auxiliary hash table structures to manage the overflowing counters at the cost of additional complexity. Figure 2 shows the schematic representation of CBF.

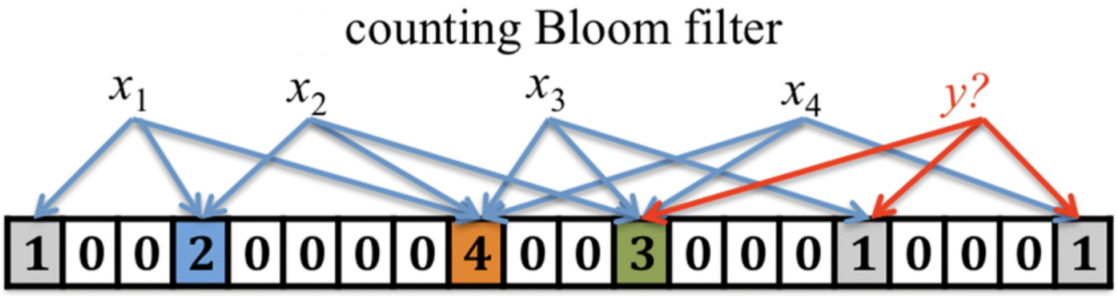

Figure 2: A counting bloom filter which includes the incremental count of items instead of simply setting the bits for accommodating the deletion operation over SBF

Blocked Bloom filters [4] (BBF):
BBF can be considered as a cache-efficient variant of SBF. The idea was to improve upon the cache-inefficiency of SBF over positive queries. The proposed way of improving was to avoid mapping over the entirety of the array of bits and instead map to a small block of bits. Essentially, checking bits within a limited range. This idea helped not only to improve the cache efficiency but also the processing time, as there was no need to go for multiple data loads in an array. Additionally, the partitioning idea set the stage for exploiting fast and efficient implementation strategies such as bit packing, multiplexing, and multi-blocking [4, 5]. Like SBF, conventional BBF too doesn't support deletion and additionally relies on a single hash function. Several implementation variants are introduced [5] to handle the partitioning-induced imbalance between different memory blocks. Some approaches [6] put forward a time-efficient improvement over the BBF by suggesting usage of a higher number of hash functions to fit cleanly alongside reinforcing the notion of a split within each block. The proposed approach helped to mitigate the processing cost and the associated split-partitioning drastically penalizes the false positive rates. However, efforts are being made [6] to mitigate the poor false positive rates by individually working towards making the localized SBFs more efficient. Figure 3 illustrates the schematic overview of the BBF in a forensic-based application.

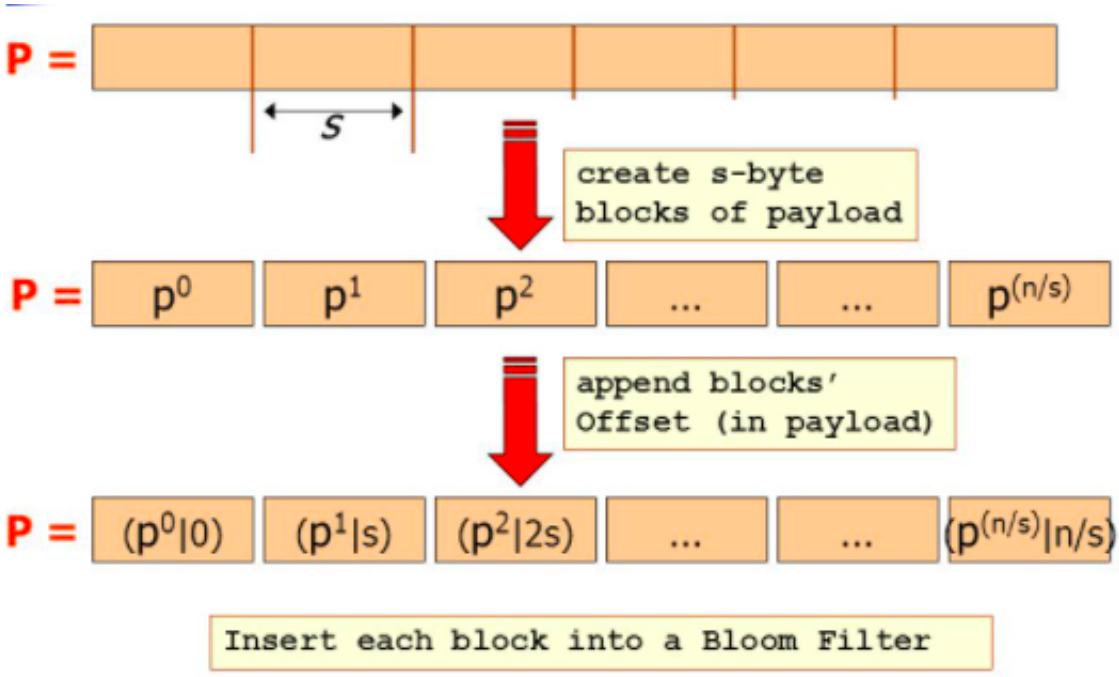

Figure 3: Blocked bloom filter demonstrating the block split on payload digest (P) bit strings to verify payload attributions in forensic applications.

Cuckoo filter [7, 8](CF):
Cuckoo Filter is presented as a light-weight probabilistic data structure for better space utilization and speedier deletion. The Cuckoo filter, unlike Bloom filters, keeps item fingerprints in two candidate buckets directly. If both candidates are fully occupied, CF removes a fingerprint from one of the candidate buckets to make room for the next item, while relocating the victim to its alternate bucket. When no more victims are triggered, the process is considered successful; however, if the number of such re-allocations exceeds a certain threshold, it is considered unsuccessful. Figure 4 provides the schematic representation of re-allocation of item x before and after inserting in the cuckoo table and the corresponding updation process of the item fingerprint signatures in the context of CF.

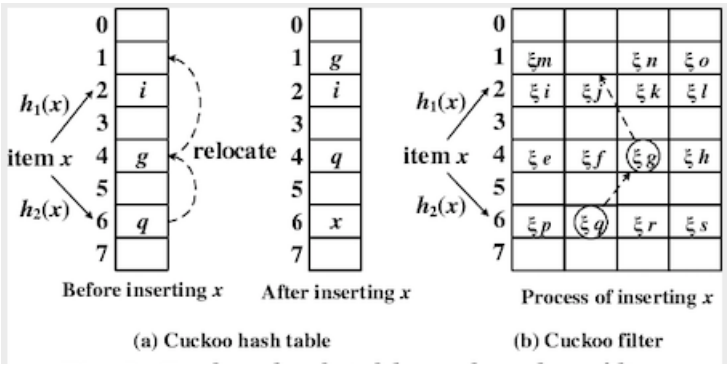

Figure 4: Re-allocation of item x before and after inserting in the cuckoo table and the corresponding updation process of the item fingerprint signatures in the context of CF

CF variants: Several efforts have been made in the literature to further enhance the performance of CF in various aspects such as false positive rate [9], flexibility [10], key-set extension [11], theoretical guarantees [12] and lookup performance [13, 14]. The state-of-the-art Cuckoo filters referred, on the other hand, rely heavily on the reallocation process that we discussed to maximize the space usage, which makes item insertion time-consuming, especially when the filter is almost full. Such designs are unsuitable for insertion-intensive applications where items are very often

added and removed. Some of the efforts [15, 16] have been made in the literature to ease the process of reallocation and to enhance the CF to be insertion-friendly while minimizing the space usage.

- **Adaptive cuckoo filter [9]:** Specifically to handle the false positive rates in the context of the conventional CF.
- **Consistent cuckoo filter [10]:** Specifically to handle the flexibility of the conventional CF.
- **Dynamic cuckoo filter [11]:** Specifically to handle the key-set extension for accommodating larger entries in the context of the conventional CF.
- **The simplified cuckoo filter [12]:** Specifically provided to improve upon the theoretical bounds of the whole re-allocation thresholds on the cuckoo tables in the conventional CF.
- **Vacuum filter & Morton filters [13, 14]:** Specifically to handle/improve the lookup throughput in the context of the conventional CF.

Summary & Motivation: Majority of the approaches discussed above are alternative implementations to improve upon the (i) false positive rate (ii) space efficiency (iii) support for deletion operations and generally proceed by either static or dynamic implementations either by using auxiliary data structures or by invoking multi-hashing strategies (using definite set of pre-determined number of hash functions of chosen behaviour, split-invoke hash functions on selected blocks or at desired locations or a desired levels of intermittent steps of insertions, rehashing & reallocating strategies) to improve upon the insertion capacity of the existing bloom filters with a common goal of achieving a low false positive rate.

The current approaches proceed with insertion by invoking either a single hash function or two hash functions by re-allocating the class of chosen set of pre-defined k-hash functions. In this work, we put forward a modified variant on the SBF, which considers insertion by item self-navigating over a k-length path of hashing at each bucket of the bloom filter. This approach will be different from SBF in the sense, the k hash functions are dynamically decided as a path by each item while self-navigating across the different hash functions in the bloom filter. More details are provided in the following section.

3 Idea methodology & Improvements

The improvement proposed can be understood as an extension of standard bloom filter with a choice of k hash functions decided dynamically by the item on navigating across the bloom filter. The overall idea is to generate a path of k-length in the bloom filters and set the bits in the path for inserting an item in the filter. The path is dynamically decided by k-hash functions which are picked randomly over n hash functions where n is the size of the bloom filter. The choice of k-hash functions is more item-personalized and hence the proposed improvement is named item-personalised bloom filter.

Working methodology: Let n be the size of the bloom filter. Instead of fixed set of k-hash functions, we are going to initialize n-hash functions equivalent to the size of the filter. Each slot of the filter holds a hash-function and a bit which is either set or not set (1 or 0). Over the n-hash functions, the input element x chooses k hash functions as its path. So, for every distinct element x, we get k number of hash-functions which are sequentially mapped as a path. This allows us to generate $C(n, k)$ unique combinations (distinct paths that can be explored to set). For each distinct x, we get one of those combinations as an output path.

Intuition: The intuition behind the idea is that when we allow the item to map sequentially in k-path we are allowing the item to choose its hash function from a set of n hash functions. This should reduce the false positive rate since we are not only setting the bits in the bloom filter based on k number of hash functions, we are also picking the k hash functions from a pool of hash functions. This should also increase the space efficiency for a set false positive rate.

Insertion and Querying: For insertion, the item is first hashed with the mother hash function h_0 and it gets mapped to a particular location in the bloom filter. If the bit is not set, we set that bit. Then, we re-hash the item using the hash function of the landed location from previous hash function. Now, the item gets re-hashed to another location in our filter and then sets that bit too. We repeat this process k times creating a path of size k. We set k-bits for an item x in the filter of size n. Figure 5 presents the schematic illustration of the proposed methodology.

For querying, similar to insertion, we trace the path based on the hash functions and the first time we hit a 0, we stop and return *item not present*, otherwise we travel till the end of our path.

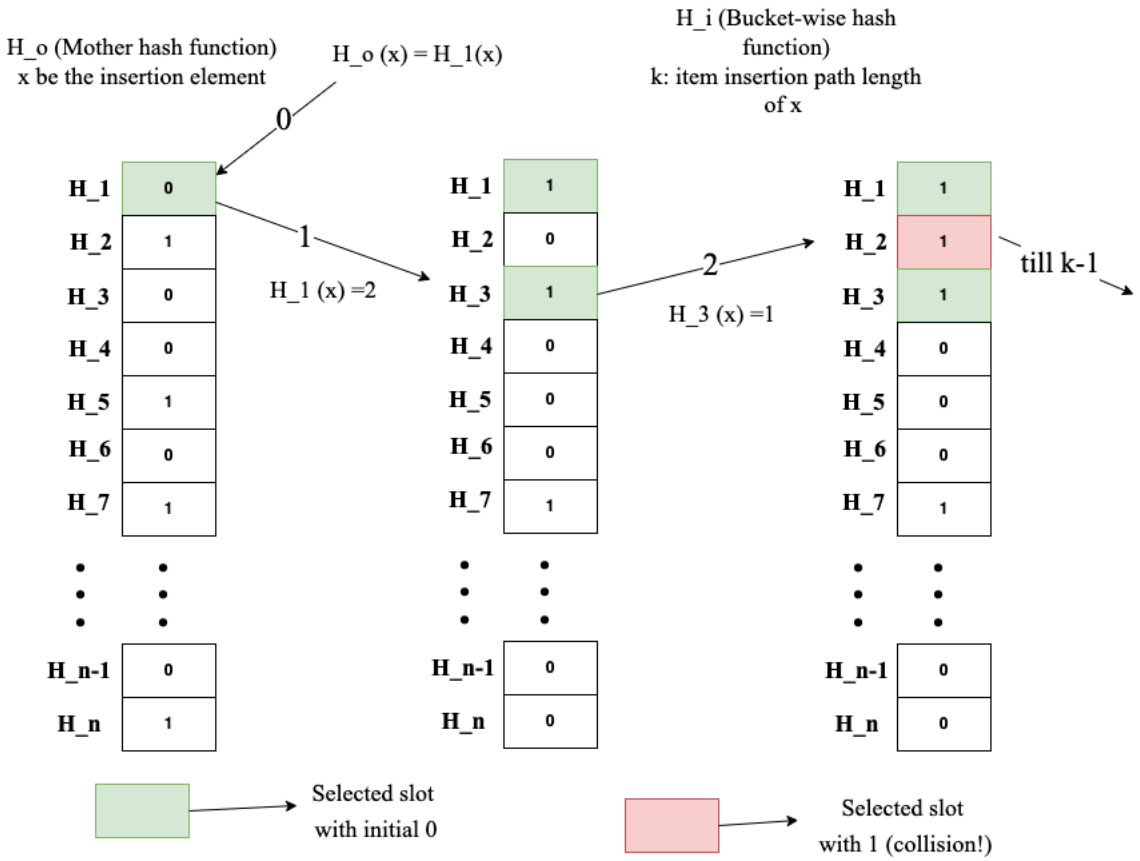

Figure 5: Schematic representation of self-navigating k length insertion path of item x in the proposed Item-personalized bloom filter(IBF)

Working variants on the proposed idea: The following provides details on possible variants that can be considered under the proposed methodology.

1. **Support for counting and deletion:** This is an extension of the above item-personalised bloom filter in the context of counting bloom filter. Instead of having bits, we can always proceed to store integers to maintain counts in the slots of the filters. This would allow us to perform counting and deleting operations. This variant can emerge as as an Item-personalised counting bloom filter.
2. **Insertion rejection:** Another variant proposed is to introduce *insertion rejection* feature. We could set up a threshold of t, and when we insert an element while tracing the k-path, if we identify any t consecutive slots in the path which are already set then we can add it to a rejection list. This variant can emerge as Item-personalized bloom filter with *i*nsertion rejection which might be of help specifically in the situations of large-scale insertions while handling high collision rates.

Big-O calculations:

- **Item-Personalised bloom filter:** For insertion, we are tracing k-paths and setting k bits in total, hence insertion is fixed to $O(k)$. For querying, we are tracing k-paths and checking k bits in total (in worst case scenario) , hence querying is bounded by $O(k)$.
- **Item-Personalised counting bloom filter:** Similar to Item-Personalised bloom filter, the insertion and querying is bounded by $O(k)$. The counting and deleting operations are also needed to be passed through the entire k-path, hence it is also bounded by $O(k)$
- **Item-Personalised bloom filter with insertion rejection:** Similar to Item-Personalised bloom filter, the insertion and querying is bounded by $O(k)$.

4 Empirical Evaluations

Experiment Details: We conducted an empirical experiment to study the performance of Item personalised bloom filter (IBF) in comparison with standard bloom filter (SBF). We took 10 different ratios of m and n beginning from $1:1$ to $1:10$ with an increment of 1 at each stage. We generated 10 different sets of random integers and for each set we produce an insertion set and a query set. The query set was designed in such a way that it consists of 10% of data from the insertion set and 90% data outside the insertion set. We ran a total of 100 trails wherein each set is inserted and queried over the 10 different ratios of m and n and then for each trial, we calculated the winner data structure. The winner was decided by the notion as *which bloom filter (IBF or SBF) generated the least false positive rate.*

Experiment Result: We see that the IBF out-performed SBF by 13.79%. We also observe that there were a lot of draws between the IBF and SBF. In the overall experiment we see that there was about 41.44% draws observed and also there were several times the SBF outperformed the IBF. Still, we need rigorous trails with multiple parameter variations to significantly state that IBF is performing better. We also observe that whenever the IBF is outperforming SBF, the difference between the min value of IBF is considerable less than the min value of SBF – making it more interesting to explore further in this direction. Figure 6 and Figure 7 show one of the above discussed sub-experiments showcasing the performance difference between SBF and IBF with 10 fixed $m:n$ ratios. We could see overall IBF showing lower FPR in most cases.

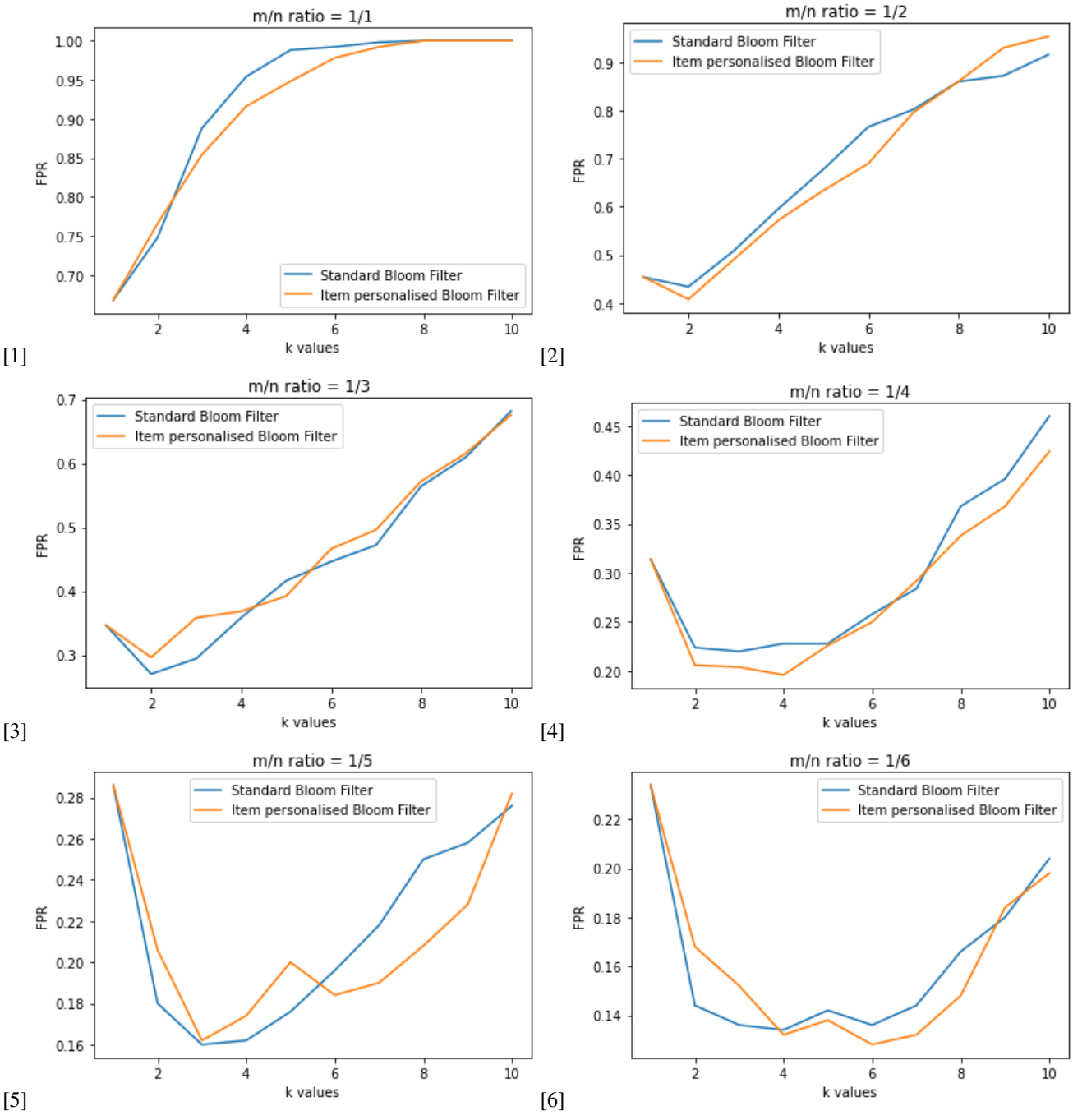

Figure 6: Plots-Part1: [1]-[6] shows the performance of IBF (proposed Item-personalized bloom filter versus SBF (standard bloom filter) over 10 different instances of m/n while varying k where m is the number of insertions, n is the size of bloom filters and k is the chosen pre-defined hash functions in SBF case and item-personalized insertion path length in IBF case

5 Conclusions and Discussion

We propose an alternative variant of bloom filter which can be considered as an extension of SBF with prioritizing dynamic k-path item-driven self-navigation based insertions. From the conducted empirical evaluations of diverse proportions of insertions, we show that the proposed IBF can marginally improve the space-efficiency while mitigating the number of false positives under a selected choice of parameters on n, m, k. IBF is out-performing the SBF at a

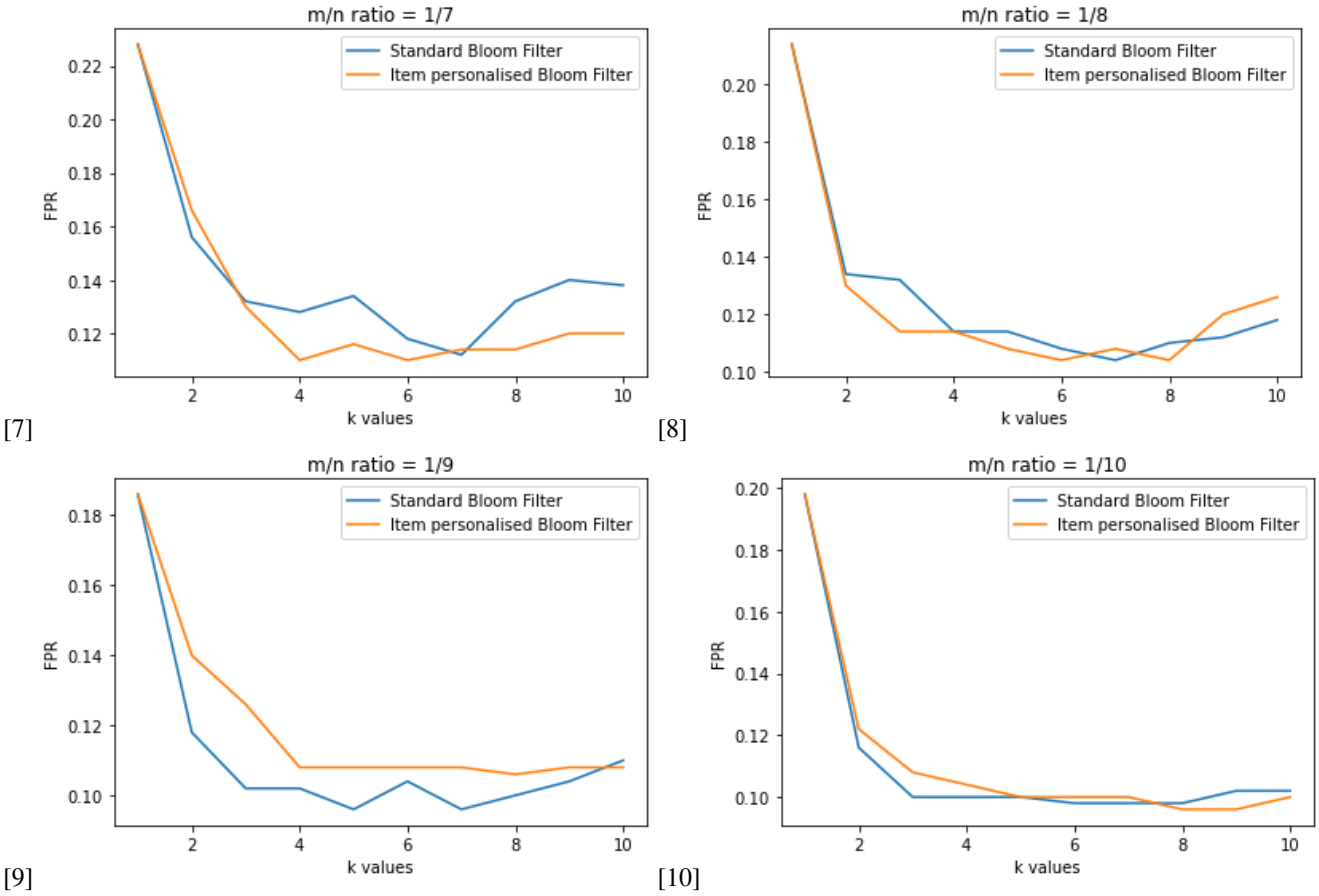

Figure 7: Plots-Part2: [7]-[10] shows the performance of IBF (proposed Item-personalized bloom filter versus SBF (standard bloom filter) over 10 different instances of m/n while varying k where m is the number of insertions, n is the size of bloom filters and k is the chosen pre-defined hash functions in SBF case and item-personalized insertion path length in IBF case

proportionally marginal value on different random trials on diverse choices of insertion sizes, hash functions and bloom filter sizes. However, we believe extensive empirical evaluations with diverse randomized data-sets, proportionally adjustable quantities of m, n, k on randomized draws of larger data-sets will further help in significantly establishing the imapct of IBF. Also, in this work we limited our comparison with SBF, but we believe this item-driven self-navigated insertions using k-length hash functions can still be verified in the context of alternative bloom filter variants.

Contributions Background research: Contributed equally in collecting research papers and summarizing works; discussed all collected literatures.

Idea analysis: Both of us discussed ideas and refined them

Evaluation: Divided the experimental codes into two segments and each of us did one segment each

Write-up: Contributed equally.

References

[1] RA Leo Elworth, Qi Wang, Pavan K Kota, CJ Barberan, Benjamin Coleman, Advait Balaji, Gaurav Gupta, Richard G Baraniuk, Anshumali Shrivastava, and Todd J Treangen. To petabytes and beyond: recent advances in probabilistic and signal processing algorithms and their application to metagenomics. *Nucleic acids research*, 48(10):5217–5234, 2020.

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

[3] 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.

[4] 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.

[5] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials*, 21(2):1912–1949, 2018.

[6] Anes Abdennebi and Kamer Kaya. A bloom filter survey: Variants for different domain applications. *arXiv preprint arXiv:2106.12189*, 2021.

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

[8] 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.

[9] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters, 2020.

[10] 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. IEEE, 2019.

[11] 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. IEEE, 2017.

[12] David Eppstein. Cuckoo filter: Simplification and analysis. *arXiv preprint arXiv:1604.06067*, 2016.

[13] Minmei Wang and Mingxun Zhou. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment*, 2019.

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

[15] 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. IEEE, 2021.

[16] Sai Medury, Amani Altarawneh, and Anthony Skjellum. Design and evaluation of cascading cuckoo filters for zero-false-positive membership services. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1061–1065. IEEE, 2021.