Extending the Applicability of Bloom Filters by Relaxing their Parameter Constraints

Paul Walther[0000−0002−5101−5793], Wejdene Mansour[0009−0008−4362−2092], Johann Maximilian Zollner[0000−0003−3742−8468], and Martin Werner[0000−0002−6951−8022]

TUM School of Engineering and Design
Technical University of Munich, Munich, Germany
{paul.walther,wejdene.mansour,maximilian.zollner,martin.werner}@tum.de

Abstract. Bloom filters serve as an efficient probabilistic data structure for representing sets of keys. They allow for set membership queries with no false negatives and with the right choice of the main parameters – length of the Bloom filter (BF), number of hash functions used to map an element to the array's indices, and the number of elements inserted – the false positive rate is optimized. However, the number of hash functions is constrained to integer values, and the length of a BF is usually chosen to be a power of two to allow for efficient modulo operations using binary arithmetic. In this paper, we relax these constraints by proposing the Rational Bloom filter, which allows for non-integer numbers of hash functions. This results in optimized fraction-of-zero values for a known number of elements to be inserted. We further enhance this with the Variably-Sized Block BF to allow for a flexible filter length, especially for large filters, with efficient computation.

Keywords: Bloom Filter · Key-Value-Store · Filter Length · Hash Function.

1 Introduction

Key-value stores proved themselves as an efficient storage model to support all steps in the data life cycle [9, 27]. Since large, sparse, and low-cardinality data can be modelled as a set and the access is often random, the requirements for an efficient in-memory representation are similar to those of a Key-Value Store. In this context, the Bloom filter (BF) was proposed as a data structure since it allows for a trade-off between its memory footprint and its error rate [19, 26].

The BF is a data structure for the efficient probabilistic storage of sets [4]. It is a binary array with methods for storing information in the filter and querying for set membership. An empty BF is an all-zero bit array of length m. Due to its structure, the BF guarantees `true` for inserted items (no false negatives) but may falsely report uninserted items to be `true` (false positives) [7, 26]. To *insert* an element x, the element is hashed with k uniformly distributed pairwise independent hash functions H_i, with $i \in \{1, \ldots, k\}$. The hashing maps from the input space of all elements (universe) to the integer range $\{0, \dots, m-1\}$. The BF is then set to `1` at the locations denoted by the same hash functions H_i. If the given value is already `1`, it remains unchanged. For the *membership query*, the BF checks the k indices computed by H_i. If any of the denoted values is `0`, the element is not part of the stored set. Otherwise, if all values are `1`, the element is present, or it is a false positive error [14]. Given n elements to store, the false positive rate p_{FP} of the BF can be calculated as [21]:

$$p_{\mathrm{FP}} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-kn/m}\right)^{k}. \tag{1}$$

The optimal number of hash functions k^* minimizes p_{FP} with a fraction of zeros $foz = \frac{1}{2}$. This maximizes the entropy of the filter and holds the highest information density, yielding

$$k^* = \frac{m}{n} \ln 2. \tag{2}$$

Nayak and Patgiri explain five main challenges with existing BF approaches: to *reduce the* p_{FP}, the *length adaption* of BFs to initially unknown dataset sizes, the *deletion of elements* without recalculation of the whole index, to implement an *efficient hashing* method without negatively influencing the performance, and the *correct determination* of k [22]. An approach to these challenges is the relaxation of the constraints on the parameters m, k, and n.

In the literature, k is restricted to integers. This reduces flexibility in constructing the optimal filter for given m and n, and especially for small k, this may result in a non-optimal p_{FP}. Figure 1 illustrates how an unconstrained, optimal choice of k can lead to improved p_{FP}. Furthermore, p_{FP} is unevenly distributed across elements in multi-hash BFs [1]. As for hash collisions, similar slots denoted by different hash functions for one element increase false positives, and to avoid this effect, using fewer hash functions is beneficial.

General hash functions h_i have to be modified to specific hash functions H_i, mapping the input space to indices of the BF $\{0, \dots, m-1\}$. The simplest method is to take the modulo $h_i \% m$. It is most efficient for $m = 2^c, c \in \mathbb{N}_0$ as it simplifies to a binary AND ($\wedge$) operation [10]:

$$H_i(x) = h_i(x) \mod m = h_i(x) \wedge (2^c - 1) \tag{3}$$

Still, in computer systems, which allocate memory in power of two increments, this restriction may have downsides if, for example, some space is already used by the operating system. Then, the BF may have at most half of the available memory size. Consequently, freely choosing m might improve the scalability and while keeping the efficient modulo operations [1, 10].

In this paper, we propose methods to extend the applicability of Bloom filters based on visions from [25]. First, we introduce a method for a **rational number** instead of only an integer number **of hash functions** k, which decreases false positive rates if the set is immutable, as it allows for more tailored BFs. Further, we propose a method to choose **non-power-of-two sizes** m **for the**

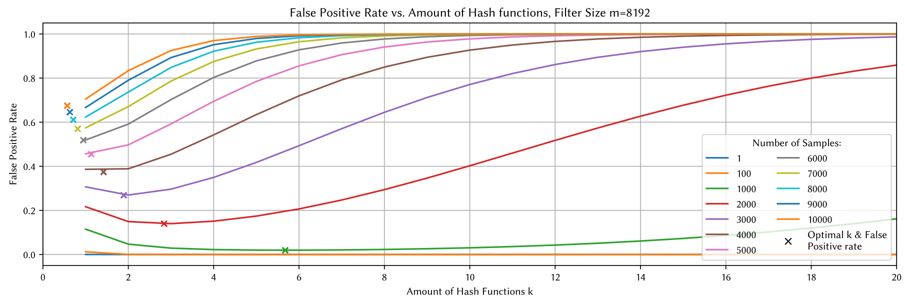

Fig. 1. Dependency of the false positive rate for a fixed filter size of $m = 8192$ on varying numbers of hash functions k and number of samples. The calculated optimal configuration for each number of samples is marked with a cross.

BF without performance degradation (by non-uniformity of access) and still without explicit modulo computations.

2 Rational Bloom Filter

For BFs, the theoretically optimal number of hash functions k^*, denoted by Eqn. 2, is generally not an integer but a rational number. Still, traditional BFs allow for an integer number of hash functions only, requiring an approximation of the optimal k^*. Allowing a non-integer k^* for a BF with known n and a constrained m (e.g., by hardware) poses the advantage to improve the false positive rate and allows for more flexibility in selecting other BF parameters. Therefore, we propose a probabilistic approach to realize rational k^*.

Definition 1. *A hash function is* ***probabilistically activated*** *if it is not applied to every sample, but instead only activated with a probability of activation* $0 \leq p_{activation} \leq 1, p_{activation} \in \mathbb{R}$.

Based on this and the knowledge that the applied hash functions H_r with $r < \lfloor k \rfloor$ give pseudo-random information which is still deterministic for a given input sample, we develop the Rational Bloom filter (RBF), which is visualized in Fig. 2:

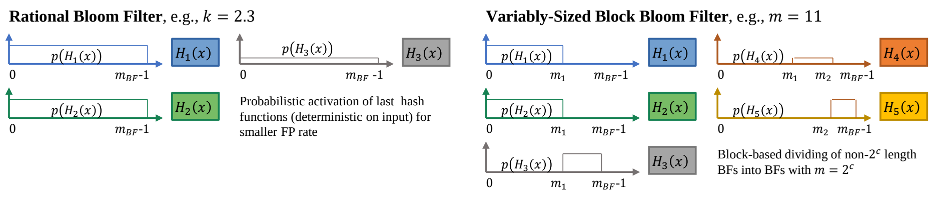

Fig. 2. Rational Bloom filter and Variably-Sized Block Bloom filter

Definition 2. *A **Rational Bloom filter (RBF)** is a BF with a non-integer number $k \in \mathbb{R}^+$ of hash functions, consisting of standard BF procedures for $\lfloor k \rfloor$ hash functions, while a hash function with probabilistic activation represents the non-integer part $k - \lfloor k \rfloor$.*

This can be implemented as described in Alg. 1, including the Hashing Trick [18].

Algorithm 1: Application of Hash Functions in the RBF

```
Data: Element $x$, BF $BF$ with length $m$, set of always-applied hash functions
      $\{H_1, \dots, H_{\lfloor k \rfloor}\}$, probabilistically activated hash function $H_{\lfloor k \rfloor+1}$,
      rational number of hash functions $k$;
Result: Set BF bits for indices denoted by the rational number of hash values
        of input element x
1 for each $H_j$ in $\{H_1, \dots, H_{\lfloor k \rfloor}\}$ do
2     Set $BF[H_j(x)] \leftarrow 1$;
3 Set $p_{activation} = k - \lfloor k \rfloor$;
4 Random hash value $H_r(x) \in [0, m]$; // No additional calculation needed if we,
    e.g., choose $H_r(x) = H_{\lfloor k \rfloor}(x)$;
5 if $H_r(x) < (p_{activation} \cdot m)$ then
6     Set $BF[H_{\lfloor k \rfloor+1}(x)] \leftarrow 1$;
```

Lemma 1. *The RBF has no false negatives.*

Proof. The given RBF with k normal and one probabilistically activated hash function $H_{\lfloor k \rfloor+1}$ has at least as many `1`-bits in the filter as the normal BF with $\lfloor k \rfloor$ hash functions since $H_{\lfloor k \rfloor+1}$ can only add `1`'s. A false negative would require a query to expect a `1` in the BF where none exists. For H_1 to $H_{\lfloor k \rfloor}$, this cannot be the case by definition of the standard BF. For $H_{\lfloor k \rfloor+1}$, activation is deterministic per input, which makes an activation during query without activation during insertion impossible.

Theorem 1. *The false positive rate p_{FP}^{RBF} of the RBF is smaller or equal to the false positive rate p_{FP}^{BF} of a normal BF: $p_{FP}^{RBF} \leq p_{FP}^{BF}$*

Proof. As described in Eqn. 2, k^* is solely determined by n and m with the assumption that the highest information can be stored for a fraction of zero $foz_{opt} = \frac{1}{2}$. For one standard BF with k_{BF} hash functions and one RBF, the RBF's k_{RBF} is chosen to be optimal, $k_{\text{RBF}} = k^*$. Based on the construction of k^* as a minimia of the false positive rate p_{FP} and the monotonicity of this function we can conclude that $p_{\text{FP}}(k^*) < p_{\text{FP}}(k_{\text{RBF}}) \forall k_{\text{RBF}} \neq k^*$. For $k_{\text{BF}} = k_{\text{RBF}}$ the optimal k^* is an integer. Therefore, $BF = RBF$ and consequently $p_{\text{FP}}^{\text{RBF}} = p_{\text{FP}}^{\text{BF}}$.

Consequently, RBFs serve as a new possibility to allow for a more flexible choice of the number of hash functions. In the following we extend this idea to variable filter lengths.

3 Variably-Sized Block Bloom Filters

For a given BF we can number each slot from 0 to $m - 1$ and this set of indices I_{BF} can be represented with J non-overlapping subsets I_{BF}^j of length $m_j = 2^c, c \in \mathbb{N}$. We can then define *subset hash functions* H_j, which only map onto their respective subset I_{BF}^j each. This can be used to efficiently compute hash functions for $m \neq 2^c, c \in \mathbb{N}_0$.

Definition 3. *The* ***Variably-Sized Block Bloom filter (VSBBF)*** *is a BF of length m_{BF}, with $2^c < m_{BF} < 2^{c+1}, c \in \mathbb{N}_0$, where the actual filter is subdivided into J blocks of sizes $m_j, j \in \{1, \dots, J\}$ with $m_j > m_{j+1}$ and $m_j = 2^c, c \in \mathbb{N}$. Each block is denoted by a set of indices I^j and is filled by k_j subset hash functions H_j^i. $k_j \in \mathbb{R}$ is thereby the optimal k_j^* for the respective filter block.*

Unlike a standard BF, the VSBBF, visualized in Fig. 2, maps hashes to blocks of length 2^c using efficient modulo operations as described in Alg. 2.

Algorithm 2: Inserting an Element in the VSBBF

```
Data: Element $x$, total elements to insert $n$, VSBBF $BF$ of total length $m_{\mathrm{BF}}$,
      uniform, pairwise independent hash functions $h_1, h_2$
Result: Inserted element $x$ into the VSBBF
binary ← BinaryRepresentation($m_{\mathrm{BF}}$);
length ← Length(binary);
hvs ← $h_1(x), h_2(x)$;
offset ← 0;
for $j$ ← 0 to length - 1 do
    if binary[j] == 1 then
        $m_j \leftarrow 2^{\mathrm{length}-j-1}$;
        $k_j \leftarrow \frac{m_j}{n} \cdot \ln 2$;
        for $i \leftarrow 0$ to $k_j$ do
            $H_i(x) = ((hvs_1 + (i + m_j) \cdot hvs_2) \& (m_j - 1))$ + offset;
            $BF[H_i(x)] \leftarrow 1$;
            offset += $m_j$;
```

Lemma 2. *The VSBBF has no false negatives.*

Proof. By definition, the filter blocks are all either normal or RBFs, and thus, it follows from Lemma 1 that the Block BF has no false negatives.

Theorem 2. *The false positive rate of VSBBF is calculated with the chain rule $p_{FP}^{Block} = \prod_j p_{FP}^j \approx \prod_j \left(1 - e^{-k_j n / m_j}\right)^{k_j}$.*

Proof. For several BFs representing the same set, which is similar to hash functions only mapping to subsets of the BF, deciding whether a queried sample is in the desired set is always requires checking all applied hash functions. An item can only be false positive if these all point wrongly to a `1` element. This is equal to all subset BFs wrongly denoting the element to be in the set.

Corollary 1. *For the optimal choice of $k_j = k_j^*$, the false positive rate of the combined filters of length $m_j = 2^{c_j}$ stays the same as the single BF of size m_{BF}:*

$$p_{FP}^{Block} \approx \prod_j \left(1 - e^{-k_j n/m_j}\right)^{k_j} = \left(1 - e^{-kn/m_{BF}}\right)^k \approx p_{FP}^{BF}. \tag{4}$$

Proof. With Eqn. 2 for optimal $k_j^* = \frac{m_j}{n} \ln 2$ and $k^* = \frac{m_{\mathrm{BF}}}{n} \ln 2$:

$$\prod_j \left(1 - e^{-\ln 2}\right)^{\frac{m_j}{n} \ln 2} = \left(1 - e^{-\ln 2}\right)^{\frac{m_{\mathrm{BF}}}{n} \ln 2} \tag{5}$$

and as $\sum_j m_j = m_{\mathrm{BF}}$ by construction, for the choice of optimal k_j^* it holds $p_{\mathrm{FP}}^{\mathrm{Block}} = p_{\mathrm{FP}}^{\mathrm{BF}}$ as we can assume optimality from RBF paradigms.

Theorem 3. *The Block BF improves the equal distribution of false positives over the to-be-tested elements compared to a standard BF of the same size.*

Proof. An element is more likely to be a false positive if its footprint has fewer `1`'s due to clashing hash values, increasing the risk of overlap with existing elements in the BF. For a fully filled filter with $foz = 0.5$ the false positive rate is $p_{\mathrm{FP}} = 0.5^k$ and for o clashing hash functions it increases to $p_{\mathrm{FP}}^{\mathrm{clash}} = 0.5^{(k-o)} > p_{\mathrm{FP}}$.

For a given Block BF, the sum of applied hash functions $\sum_j k_j^*$ is equal to the optimal number k^* for a filter of the summed lengths of the blocks $m = \sum_j m_j$. The probabilities for one element x being inserted into a filter BF with k^* hash functions having less than k^* hash values are

$$p_{\mathrm{clash}} = \sum_{i=1}^{k-1} \frac{i}{m_{\mathrm{BF}}}, \qquad p_{\mathrm{clash}}^{\mathrm{Block}} = \sum_j \sum_{i=1}^{k_j-1} \frac{i}{m_j}. \tag{6}$$

After reformulation, we can show for all $J > 1$ that $p_{\mathrm{clash}} > p_{\mathrm{clash}}^{\mathrm{Block}}$.

Corollary 2. *The proposed solution requires no storage overhead for the description of the block sizes m_j and the number of hash functions k_j.*

Proof. This information is contained in the non-blocked representation's properties, the assumption that blocks are sorted descendingly with $m_{j+1} < m_j$ and the rule that the optimal number of hash functions k_j^* is always chosen per block.

The proposed solution reduces processing time: $t(BF) \geq t(BF^{\mathrm{Block}})$ for insertion and querying, when $m_{\mathrm{BF}} \neq 2^c, c \in \mathbb{N}_0$ and $\sum_j m_j^{\mathrm{Block}} = m_{\mathrm{BF}}$ and the runtime of one modulo calculation by $m_{\mathrm{BF}} \neq 2^c$ is computationally more expensive than the computation of J modulo operations with the binary bit trick [24].

A further advantage of Block BF is the easy and meaningful compression by simply taking subsets of the filter, which comprise a certain number of blocks. In tendency, this enables more compression steps than the standard BF of size $m_{\mathrm{BF}} = 2^c$ as not only halving the size is possible but also any combination of block sizes applied in the BF. Available compression ratios are $\frac{\sum_i m_i}{m_{\mathrm{BF}}}$, with $i \subseteq j$.

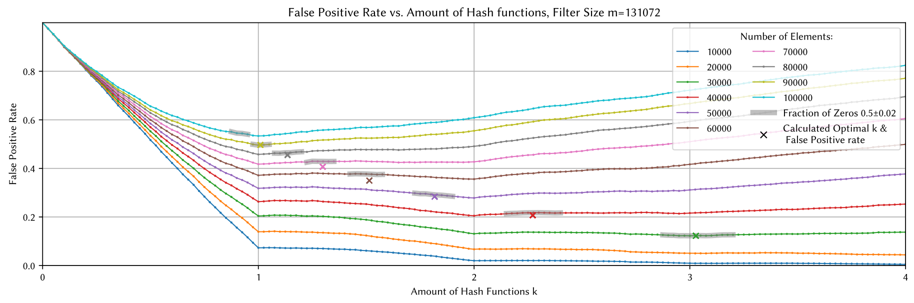

Fig. 3. False positive rate of RBFs with filter size 131,072 for different k and n, queried on 10,000 unseen elements. Experiments with *foz* $= 0.5 \pm 0.02$ are shadowed grey. Theoretically optimal configurations are denoted by a cross.

4 Implementation Artifacts

To validate our theories, we implemented RBF and VSBBF in C++ with a Murmur Hash using the hashing trick and efficient modulo calculation with binary arithmetic. As a baseline, we also implemented a standard BF with the same optimizations. Interestingly, *RBF* showed unexpected artifacts: The false positive rate has local minima not at the theoretically optimal rational k^*, but near integer values instead, as shown in Fig.3. For example, with $m = 131,072$ and $n = 60,000$, the global minimum occurs at $k = 2$ and another local minimum at $k = 1$, although the optimal value lies in between. This was counterintuitive, and we cannot yet fully explain it. We suspect this is due to the non-uniform behavior of the hashing algorithm. However, similar artifacts were also observed with SHA256, which should lead to a more uniform distribution of hash values. Additionally, larger filter sizes and numbers of elements inserted did not mitigate this, though logically, this should reduce the impact of non-uniform hash functions. Comparing the *foz* of the near-optimal RBF and BFs with integer k shows that RBF achieves better *foz* rates. For *VSBBF*, the false positive rates are similar to a standard BF, but no insertion time improvements were observed. Although our practical results fall short of the theoretical gains, we still believe our methods are a valuable contribution to BF theory and open new application possibilities, especially for learned BF approaches and input-dependent hash functions.

5 Related Work

To the best of our knowledge, no prior work addresses rational numbers of hash functions, though many improve the hashing: *Perfect hashing* uniformly maps inputs to m buckets [14]. Common hash functions like Murmur Hash [2] are used with consecutive modulo operations. For $k > 2$, *double hashing* calculates two hashes and combines them $h_i(x) = h_1(x) + f(i)h_2(x)$ [14, 18]. *Partitioned hashing* [14] assigns each function disjoint ranges of length m/k, which may increase false positives due to more `1`s. Hao et al. group inputs and apply *different uniform hash functions*, optimizing for the highest *foz* [17]. Bruck et al. allocates hash functions based on query likelihood [7]. In [27], *non-uniform hash functions* and float-based representations on GPUs come at the cost of higher memory and complexity; unlike our RBF, which only relaxes parameter constraints. Several BFs adapt their size to the number of elements. *Incremental BF* [15] sets a fill threshold for a minimum *foz* and adds new filters when reached. Similarly, [1] proposes adding *plain BFs* of increasing size $m_i = m_0 \cdot s^{l-1}$, applying a geometric progression on error bounds to keep false positives and elements-per-filter constant. *Slicing* assigns distinct BF regions per hash function to avoid overlap and achieves more uniform false positives [1, 6, 8]. *Dynamic BF* [13] supports deletions and is adopted from Counting BFs [5, 12] and *Block BF* [23], which consists of multiple small cache-line-sized BFs and inserts each element in just one. Furthermore, *Combinatorial BF* and *Partitioned Combinatorial BF* [16] use a set of BFs when one element belongs to several sets and partition a BF into smaller ones of similar size. These approaches split the BF into smaller pieces for scalability or memory locality, but still use power-of-two sizes and similarly-sized filter subparts. Last, *learned BFs* mimic BFs with a learned function, allowing false negatives and using a small BF to filter them out [20]. Beyond direct improvements to BFs, *Quotient* [3] and *Cuckoo Filters* [11] enhance data locality and space efficiency, and support deletions without recalculations, often outperforming BFs. Still, BFs variants remain popular in data management systems for their simple architecture. Our approaches allow for improvements in these domains without a general change in system architecture.

6 Conclusion

By relaxing the classic constraints on Bloom filter parameters, such as requiring a natural number of hash functions and filter sizes that are powers of two, we present two new BF designs. The RBF uses a probabilistic activation of hash functions, which is still deterministic with respect to the inputs to mimic the theoretically optimal number of hash functions. We prove that this theoretically achieves superior false positive rates compared to standard BFs of the same size. Further, VSBBFs allow for variable filter sizes and efficient modulo operation, enabling a better distribution of false positives over all elements in the universe, as a clash of hash functions is less probable. The probabilistic activation of hash functions in RBFs opens new research directions, e.g., how learned approaches might use rational numbers of hash functions. This extends existing approaches that only rely on learned pre-filters to increase learned BF's efficiency.

Acknowledgements

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 507196470.

Bibliography

[1] Almeida, P.S., Baquero, C., Preguiça, N., Hutchison, D.: Scalable Bloom Filters. Information Processing Letters **101**(6), 255–261 (2007). https://doi.org/10.1016/j.ipl.2006.10.007

[2] Appleby, A.: aappleby/smhasher: MurmurHash: GitHub Repository (2008), https://github.com/aappleby/smhasher

[3] Bender, M.A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B.C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R.P., Zadok, E.: Don't thrash: How to Cache your Hash on Flash. Proceedings of the VLDB Endowment **5**(11), 1627–1637 (2012). https://doi.org/10.14778/2350229.2350275

[4] Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970). https://doi.org/10.1145/362686.362692

[5] Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., Varghese, G.: An Improved Construction for Counting Bloom Filters. Lecture Notes in Computer Science, vol. 4168, pp. 684–695. Springer Berlin Heidelberg (2006). https://doi.org/10.1007/11841036_61

[6] Bose, P., Guo, H., Kranakis, E., Maheshwari, A., Morin, P., Morrison, J., Smid, M., Tang, Y.: On the false-positive rate of Bloom filters. Inf. Process. Lett. **108**(4), 210–213 (2008). https://doi.org/10.1016/j.ipl.2008.05.018

[7] Bruck, J., Gao, J., Jiang, A.: Weighted Bloom Filter. In: 2006 IEEE International Symposium on Information Theory. pp. 2304–2308. John Wiley (2006). https://doi.org/10.1109/ISIT.2006.261978

[8] Chang, F., Wu-chang Feng, Kang Li: Approximate caches for packet classification. In: IEEE INFOCOM 2004. pp. 2196–2207. IEEE (2004). https://doi.org/10.1109/infcom.2004.1354643

[9] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. ACM SIGOPS Operating Systems Review **41**(6), 205–220 (2007). https://doi.org/10.1145/1323293.1294281

[10] Estébanez, C., Saez, Y., Recio, G., Isasi, P.: Performance of the most common non-cryptographic hash functions. Software: Practice and Experience **44**(6), 681–698 (2014). https://doi.org/10.1002/spe.2179

[11] Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo Filter. In: Proceedings of the 2014 CoNEXT: December 2-5, 2014, Sydney, Australia, pp. 75–88. ACM (2014). https://doi.org/10.1145/2674005.2674994

[12] Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache. ACM SIGCOMM Computer Communication Review **28**(4), 254–265 (1998). https://doi.org/10.1145/285243.285287

[13] Guo, D., Wu, J., Chen, H., Yuan, Y., Luo, X.: The Dynamic Bloom Filters. IEEE Transactions on Knowledge and Data Engineering **22**(1), 120–133 (2010). https://doi.org/10.1109/TKDE.2009.57

[14] Gupta, D., Batra, S.: A short survey on bloom filter and its variants. In: 2017 International Conference on Computing, Communication and Automation (ICCCA). pp. 1086–1092 (2017). https://doi.org/10.1109/CCAA.2017.8229957

[15] Hao, F., Kodialam, M., Lakshman, T.V.: Incremental Bloom Filters. In: INFOCOM 2008. The 27th Conference on Computer Communications. IEEE. pp. 1067–1075. IEEE Computer Society (2008). https://doi.org/10.1109/INFOCOM.2008.161

[16] Hao, F., Kodialam, M., Lakshman, T.V., Song, H.: Fast Multiset Membership Testing Using Combinatorial Bloom Filters. In: IEEE INFOCOM 2009. pp. 513–521 (2009). https://doi.org/10.1109/INFCOM.2009.5061957

[17] Hao, F., Kodialam, M., Lakshman, T.V.: Building high accuracy bloom filters using partitioned hashing. In: Proceedings of the 2007 ACM SIGMETRICS. pp. 277–288. ACM Digital Library (2007). https://doi.org/10.1145/1254882.1254916

[18] Kirsch, A., Mitzenmacher, M.: Less Hashing, Same Performance: Building a Better Bloom Filter. In: Algorithms u2013 ESA 2006, Lecture Notes in Computer Science, vol. 4168, pp. 456–467. Springer Berlin Heidelberg (2006). https://doi.org/10.1007/11841036_42

[19] Lu, G., Nam, Y.J., Du, D.H.C.: BloomStore: Bloom-Filter based memory-efficient key-value store for indexing of data deduplication on flash. In: 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies. pp. 1–11. IEEE (2012). https://doi.org/10.1109/MSST.2012.6232390

[20] Mitzenmacher, M.: A Model for Learned Bloom Filters and Optimizing by Sandwiching. In: Advances in Neural Information Processing Systems. vol. 31. Curran Associates (2018), https://proceedings.neurips.cc/paper_files/paper/2018/file/0f49c89d1e7298bb9930789c8ed59d48-Paper.pdf

[21] Mullin, J.K.: A second look at bloom filters. Communications of the ACM **26**(8), 570–571 (1983). https://doi.org/10.1145/358161.358167

[22] Nayak, S., Patgiri, R.: A Review on Role of Bloom Filter on DNA Assembly. IEEE Access **7**, 66939–66954 (2019). https://doi.org/10.1109/ACCESS.2019.2910180

[23] Putze, F., Sanders, P., Singler, J.: Cache-, hash-, and space-efficient bloom filters. ACM Journal of Experimental Algorithmics **14** (2009). https://doi.org/10.1145/1498698.1594230

[24] Reed, I.: A class of multiple-error-correcting codes and the decoding scheme. Transactions of the IRE Professional Group on Information Theory **4**(4), 38–49 (1954). https://doi.org/10.1109/tit.1954.1057465

[25] Walther, P.: Advancements of Randomized Data Structures for Geospatial Data. In: Proceedings of the Workshops of the EDBT/ICDT 2024 (2024), https://ceur-ws.org/Vol-3651/PhDW-1.pdf

[26] Werner, M.: GloBiMapsAI: An AI-Enhanced Probabilistic Data Structure for Global Raster Datasets. ACM Transactions on Spatial Algorithms and Systems **7**(4), 1–24 (2021). https://doi.org/10.1145/3453184

[27] Werner, M., Schönfeld, M.: The Gaussian Bloom Filter. In: Database systems for advanced applications, LNCS, vol. 9049, pp. 191–206. Springer (2015). https://doi.org/10.1007/978-3-319-18120-2_12